

Abstraction and Refinement Techniques for Ternary Symbolic Simulation with Guard-value Encoding

by

Bo Yang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2022

© Bo Yang 2022

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: Alan J. Hu
Professor, Dept. of Computer Science,
University of British Columbia

Supervisor(s): Mark D. Aagaard
Associate Professor, Dept. of Electrical and Computer Engineering,
University of Waterloo

Internal Member: Arie Gurfinkel
Associate Professor, Dept. of Electrical and Computer Engineering,
University of Waterloo
Hiren Patel
Professor, Dept. of Electrical and Computer Engineering,
University of Waterloo

Internal-External Member: Richard Trefler
Associate Professor, Cheriton School of Computer Science,
University of Waterloo

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

We propose a novel encoding called guard-value encoding for the ternary domain $\{0, 1, X\}$. Among the advantages it has over the more conventional dual-rail encoding, the flexibility of representing X with either of $\langle 0, 0 \rangle$ or $\langle 0, 1 \rangle$ is especially important. We develop data abstraction and memory abstraction techniques based on the guard-value encoding. Our data abstraction reduces much more of the state space than conventional ternary abstraction’s approach of over-approximating a set of Boolean values with a smaller set of ternary values. We also show how our data abstraction can enable bit-width reduction which helps further simplify verification problems. Our memory abstraction is applicable to any array of elements which makes it much more general than the existing memory abstraction techniques. We show how our memory abstraction can effectively reduce an array to just a few elements even when existing approaches are not applicable. We make extensive use of symbolic indexing to construct symbolic ternary values which are used in symbolic simulation. Lastly, we give a new perspective on refinement for ternary abstraction. Refinement is needed when too much information is lost due to use of the ternary domain such that the property is evaluated to the unknown X . We present a collection of new refinement approaches that distinguish themselves from existing ones by modifying the transition function instead of the initial ternary state and ternary stimulus. This way, our refinement either preserves the abstraction level or only degrades it slightly. We demonstrate our proposed techniques with a wide range of designs and properties. With data abstraction, we usually observe at least $10\times$ improvement in verification time compared to Boolean verification algorithms such as Boolean Bounded Model Checking (BMC), as well as usually at least $2\times$ and often $10\times$ improvement over conventional ternary abstraction. Our memory abstraction significantly improves how the verification time scales with the design parameters and the depth (the number of cycles) of the verification. Our refinement approaches are also demonstrated to be much better than existing ones most of the time. For example, when verifying a property of a synthetic example based on a superscalar microprocessor’s bypass paths, with our data abstraction, it takes 505 seconds while both of ternary abstraction and BMC time out at 1800 seconds. The bit-width reduction can further save 44 seconds and our memory abstraction can save 237 seconds. This verification problem requires refinement. If we substitute our refinement with an existing approach, the verification time with the data abstraction doubles.

Acknowledgements

Foremost, I would like to thank my supervisor, Professor Mark Aagaard for sharing his immense knowledge and experience, for mentoring me in research and life, for encouraging me through the lows, and for giving me freedom to explore in research. I can never thank enough. I would like to thank Professor Guang Gong for countless support and insightful research discussions.

I would like to thank my committee members, Professor Alan Hu, Professor Arie Gurfinkel, Professor Hiren Patel and Professor Richard Treffer for their time and valuable feedback. It has been a privilege.

Furthermore, I would like to thank Professor Kalikinkar Mandal and people I met in ComSec Lab, for the seminars, collaboration, discussions and friendship.

I would like to thank the researchers and developers behind the tool ABC and Yosys for creating such wonderful open-source tools that make it a lot easier for people to experiment with ideas.

I would like to thank people I work with during the internship at Apple for making my experience there enjoyable and fruitful. Special thanks to Antonio Caldeira, Nari Krishnamurthy and Sebastian Skalberg.

I would like to thank Wayne Yun for providing me with opportunities at AMD and sharing his deep knowledge in verification.

Last but not least, I would like to thank Kai Jiang and all my friends in Waterloo for all the supports and good time!

Dedication

To my family.

Table of Contents

List of Figures	x
List of Tables	xi
List of Codes	xiii
1 Introduction	1
2 Background and Related Work	6
2.1 Signals and States	6
2.2 Symbolic Indexing	8
2.3 Transition System	8
2.4 BMC and Inductive Proof in Ternary Domain	10
2.4.1 Bounded Model Checking	11
2.4.2 Induction	11
2.5 Ternary Abstraction	12
2.6 Ternary Encoding	15
2.7 Related Work	18
3 Guard-value Encoding	23
3.1 Guard-value Encoding	23
3.2 Comparing the Guard-value and Dual-rail Implementations of Ternary Transition Functions	25
3.3 Value States Simplification using the Care Set	28
3.4 Value Transition Function Simplification using the Care Set	31
3.4.1 Memory Abstraction by Value Transition Function Simplification	32
3.5 Summary	35

4	Effective Verification with Ternary Domain	36
4.1	Introduction	36
4.2	BMC-DR and BMC-GV	38
4.3	BMC-GV and BMC	40
4.4	Other Advantages of the Ternary Verification Based on the Guard-value Encoding	42
5	Data Abstraction	44
5.1	A Motivating Example	44
5.2	Data Abstraction with the Guard-value Encoding	46
5.2.1	Formalization	47
5.2.2	Equal to a Vector of Boolean Variables	51
5.2.3	Equal to Either of Two Vectors of Boolean Variables	53
5.2.4	Less Than a Vector of Boolean Variables	56
5.3	Summary	60
6	Guard-value Memory Abstraction	61
6.1	A Motivating Example	61
6.2	Formalization	63
6.2.1	A Candidate for Tr'	64
6.2.2	A More General Candidate for Tr'	65
6.3	Summary	66
7	Refinement	67
7.1	Comparison with Existing Refinement Approaches	69
7.2	A Complete Partially Ordered Set of Ternary Functions	71
7.3	Refinement by Re-implementing the Unrolled Boolean Transition Function	73
7.4	Refinement by Re-implementing the Boolean Transition Function	75
7.4.1	Refinement for the Less-than Data Abstraction	76
7.4.2	Discussion of the Completeness	77
7.5	Refinement by Over-approximating the Boolean Transition Function	77
7.6	Summary	79

8	Benchmarks and Results	80
8.1	Benchmarks	80
8.1.1	Content Addressable Memory	81
8.1.2	Scheduler	81
8.1.3	Response Buffer	82
8.1.4	Shared Resource Manager	84
8.1.5	Floating Point Subtraction	85
8.1.6	Bypass	86
8.2	Results	86
8.2.1	Experiments Setup	88
8.2.2	Overview of Results	90
8.2.3	CAM	97
8.2.4	Scheduler	100
8.2.5	Response Buffer	103
8.2.6	Shared Resource Manager	107
8.2.7	Floating Point Subtraction	109
8.2.8	Bypass	109
9	Conclusion	112
	References	114
	APPENDICES	119
A	Verilog Codes	120
A.1	CAM	120
A.2	RBuf	121
A.3	Bypass	122

List of Figures

2.1	An AIG representation of a multiplexer	15
2.2	An AIG representation of a ternary multiplexer using dual-rail encoding . .	18
5.1	Content-addressable memory	45
7.1	Conventional (path ‘B’) and our way (path ‘A’) of constructing an unrolled ternary implementation from a Boolean transition function	68
7.2	A different AIG implementation of a multiplexer for abstraction refinement	72
7.3	Refinement by re-implementing the unrolled Boolean transition function . .	75
7.4	Refinement by re-implementing the Boolean transition function	76
7.5	Refinement by over-approximating the Boolean transition function	79
8.1	Response buffer	83
8.2	Initial state of the FIFO storing available resources	85
8.3	Bypass design	87
8.4	A run of the bypass	87
8.5	GV is faster than DR, Roorda and BMC at proof	94
8.6	GV is faster than DR, Roorda and BMC at finding corner-case bugs	94
8.7	Effectiveness of GV memory abstraction	95
8.8	Effectiveness of GV bit-width reduction	95
8.9	Effectiveness of ternary refinement	96

List of Tables

2.1	Ternary AND ($\hat{\wedge}$) and NOT ($\hat{\neg}$)	10
2.2	The high rail and low rail of the output of a ternary AND ($\hat{\wedge}$)	16
2.3	The high rail and low rail of the output of a ternary NOT ($\hat{\neg}$)	16
3.1	The guard and value of the output of a ternary AND ($\hat{\wedge}$)	24
3.2	The guard and value of the output of a ternary NOT ($\hat{\neg}$)	24
3.3	Comparison of encodings	30
4.1	Comparing the value verification and guard verification of BMC-GV against BMC-DR and BMC	41
5.1	Ternary abstraction for the less-than predicate.	58
5.2	Ternary abstraction for the greater-than or equal-to predicate.	58
8.1	The mapping from x-axis' labels to verification problems	93
8.2	CAM (data correctness): comparison of GV, DR, Roorda and BMC	97
8.3	CAM (data correctness): effective GV memory abstraction	97
8.4	CAM (data correctness): effective bit-width reduction	98
8.5	CAM (tag uniqueness): comparison of GV, DR, Roorda and BMC	99
8.6	Scheduler: comparison of GV, DR, Roorda and BMC	100
8.7	Scheduler: comparison of the value verification using different combinations of data abstraction and encX 0 and encX symb	101
8.8	Scheduler: refinement comparison, encX 0	101
8.9	Scheduler: refinement comparison, encX symb	102
8.10	RBuf (data correctness): comparison of GV, DR, Roorda and BMC	103
8.11	RBuf (data correctness): adapting the data abstraction to BMC	104

8.12 RBuf (data correctness): effective simple memory abstraction	104
8.13 RBuf (data correctness): effective GV memory abstraction	104
8.14 RBuf (data correctness): refinement comparison	105
8.15 RBuf (ID uniqueness): comparison of GV, DR, Roorda and BMC	105
8.16 RBuf (ID uniqueness): effective bit-width reduction	106
8.17 SRM: comparison of GV, DR, Roorda and BMC	107
8.18 SRM: comparison of GV, DR, Roorda and BMC	107
8.19 SRM: effective bit-width reduction	108
8.20 SRM: comparison of GV, DR, Roorda and BMC for buggy designs	108
8.21 SRM: adapting the data abstraction to BMC	108
8.22 FPSub: comparison of GV, DR, Roorda and BMC	109
8.23 Bypass: comparison of GV, DR, Roorda and BMC	109
8.24 Bypass: effective bit-width reduction	110
8.25 Bypass: effective GV memory abstraction	110
8.26 Bypass: comparison of GV, DR, Roorda and BMC for the buggy bypass .	110
8.27 Bypass: refinement comparison	111

List of Codes

3.1	Maximum value state space reduction	29
6.1	The original value transition function for data in the i-th slot	62
6.2	The new value transition function for data in the i-th slot	62
8.1	The original value transition function for <code>req_id</code>	83
8.2	The new transition function for <code>req_id</code>	83
A.1	Without GV memory abstraction	120
A.2	With GV memory abstraction	120
A.3	Without Boolean memory abstraction	121
A.4	With Boolean memory abstraction	121
A.5	Without GV memory abstraction	122
A.6	With GV memory abstraction	122
A.7	Without Boolean memory abstraction	122
A.8	With Boolean memory abstraction	123
A.9	Without GV memory abstraction	124
A.10	With GV memory abstraction	124

Chapter 1

Introduction

Abstraction has been a successful technique in tackling the “state space explosion” problem of formal verification. Among the extensive literature on abstraction, we are particularly interested in ternary abstraction because of its ease of constructing the abstracted model, and memory abstraction which represents the techniques that target hardware structures based on arrays. In general, abstraction reduces the state space and thus helps hardware verification run faster.

In ternary abstraction, the abstraction of a hardware design can be simply constructed by substituting the Boolean gates with corresponding ternary ones. The ternary domain extends the Boolean domain with the unknown value X to abstract away unnecessary information for verification. With ternary domain, a set of Boolean states can be abstracted with ternary states: $(0, X, X)$ abstracts $\{(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1)\}$. Abstraction is done by using X , rather than a Boolean variable, on selected signals in the stimulus for verification. The state space reduction achieved with ternary states explains how ternary abstraction can speed up verification. With ternary abstraction, it is the abstracted design that is checked to see whether the property is satisfied. If the ternary design satisfies the property, it is implied that the original hardware design is correct. Symbolic Trajectory Evaluation (STE) has been a successful model checking technique for hardware verification that utilizes a ternary abstraction. Note that to be precise, STE uses a quaternary domain which has a fourth value representing conflict. A conflict value indicates problems in the STE property. We propose a novel data abstraction technique based on ternary abstraction that can reduce significantly more state space than STE. Our data abstraction reduces the state space in a way that is only made possible by our novel guard-value (GV) encoding of the ternary domain. We will describe it more in detail soon. But on the surface, it is already very different from STE. Our data abstraction uses a ternary domain instead of a quaternary domain which enables an optimization of choosing between two different encodings of X . Additionally, we do not limit ourselves to a particular property specification language like STE. Instead, we assume that the property is compiled into the design as a primary output and the task is to verify that the primary output is always 1. This setup is adopted widely in model checking literature. In this thesis, because we only introduce X at

the inputs and initial states, it is impossible for a signal to have a conflict value¹. Therefore, from a practical point of view, we can consider STE uses ternary domain as well, which allows us to compare our techniques against STE. Because the unique property language STE uses, we need to take extra steps in order for STE to solve the same verification problem as our techniques. We will detail these steps in the benchmark chapter.

Memory abstraction can also be achieved using ternary abstraction, as demonstrated by STE. Additionally, there is a Boolean approach such as the work done by Bjesse in [6]. The Boolean memory abstraction reduces the memory to just a few memory cells whose addresses are symbolic constants. These cells store all the information needed for verification. If reading from a location that is not equal to one of the symbolic constants, the abstracted memory returns an arbitrary value. In some cases, Boolean memory abstraction can reduce an enormous memory to just a few cells. However, it also has limitations, for example, it cannot abstract certain memories such as content-addressable-memory (CAM). In this thesis, we propose a new technique for memory abstraction based on ternary abstraction and made possible by GV encoding. When compared to Boolean memory abstraction, our memory abstraction is much more general and widely applicable. In theory, it can be applied to any array of elements. We show that for any memory the existing Boolean technique can abstract, our technique can achieve the same effect. When compared with memory abstraction using a conventional ternary domain in STE, our memory abstraction is more powerful and can achieve significantly more state space reduction.

New techniques for data abstraction and memory abstraction are two of the main contribution of this thesis. They significantly extend the existing work on ternary abstraction. In the implementation of these two techniques, we use off-the-shelf logic optimizations to simplify the SAT problem represented as an And-Inverter-Graph (AIG) before translating it to a conjunctive-normal-form (CNF) formula and giving it SAT solvers. With this approach, we demonstrate that there is great synergy between our abstraction techniques and logic optimizations: the time invested in the logic optimization procedures is far exceeded by the savings in SAT solving time.

Refinement is needed when too much information is lost such that the property evaluates to X. Another major contribution is to propose a novel refinement framework that does refinement by modifying the design. It is a completely different approach of refinement from the existing ones which modifies the input stimulus and initial state to make them have less X.

We benchmark our techniques against BMC, conventional STE and STE with Roorda’s encodings. We use 6 complex circuits with a total of 8 properties, inspired by real world examples. The examples range from a buffer design that is needed for processing traffic from real-world bus protocols to a synthetic design based on a superscalar microprocessor’s bypass path. Such designs have considerable complexity and are not the typical data-path designs where people apply STE.

¹Assuming the stimulus has been checked for consistency.

We now provide a high-level description of each of our contributions with focus on how the GV encoding plays a key role to enable our abstraction techniques.

Guard-value (GV) encoding The critical role GV encoding plays in our data abstraction and memory abstraction stems from its ability of dividing a ternary verification into two Boolean verification problems and its flexibility in encoding X . In the GV encoding, X can be encoded by either $\langle 0, 0 \rangle$ or $\langle 0, 1 \rangle$. The two Boolean values encoding a ternary value are called guard (field) and value (field) respectively. The GV encoding indicates that the guard is the condition under which we care about the value field, i.e., when the guard is 0, we don't care about the value field. This immediately inspires the use of don't-care based optimizations to simplify the value field. Using the GV encoding, the ternary transition function for a signal is encoded by two Boolean transition functions that compute the guard and value field of the signal's value in the next state. A signal's guard transition function is usually larger than its original transition function in terms of the number of AND gates in the AIG representation. A signal's value transition function has the same size as the original transition function. In fact, they are identical after substituting the inputs of the original transition function with signals denoting their value fields. For example, consider the transition function for an AND gate, $o = a \wedge b$. We use a_v to denote signal a 's value field. Then, by replacing a with a_v and b with b_v , we can derive the output's value transition function $o_v = a_v \wedge b_v$. A ternary verification is divided into 1) checking that the guard of the primary output is 1 (guard verification) which essentially verifies that X cannot propagate to the primary output representing the property and 2) checking that the primary output's value field is 1 (value verification) which essentially checks that the Boolean transition system satisfies the property when its input stimulus and initial state are the value fields of their ternary abstraction. The guard verification as an X propagation problem intuitively is simpler to solve considering majority of the signals' values are usually not X , which makes it invite tailored/specialized algorithms to solve it. We demonstrate that off-the-shelf logic optimizations work well in simplifying and often even solving it directly, meaning SAT solvers are not needed. As for the value verification, because it is identical to the original verification problem except that the input and initial state space of the value verification is the value field of the ternary abstraction of the Boolean input and initial state space of the original verification, a natural idea is to have as many X as possible in the ternary abstraction and use the don't-care optimizations to further reduce the cardinality of the state space's value field. This is exactly the main goal of our data abstraction and memory abstraction: they are applied to reduce the state space of the value verification so that it can outperform the original Boolean verification with the margin that must be more than enough to offset the time spent for the guard verification. This in turn provides us with a criteria to decide whether a verification problem is suitable for our abstraction techniques: can our abstraction techniques reduce enough of the value field of the input and initial state space to make the value verification significantly outperforms the Boolean verification approaches?

Data abstraction Our data abstraction uses user-provided hints regarding the least information about some input signals or registers' values needed for verification to abstract

their state space with ternary states. Then, don't-care optimizations are used to reduce the cardinality of these signals' value field state space while balancing the size of value state space's symbolic representation as an AIG. The data abstraction is applied to the input stimulus and initial states which will result in reduced state space for other signals at their fan-out. In Chapter 5, we will formalize aforementioned process and state the priority of the optimization goals. Additionally, to make it easier to use, we will show how our data abstraction works for a variety of common scenarios, such as when we only care about 1) whether a vector of signals is equal to a certain (symbolic) value; 2) whether a vector of signals is smaller (or larger) than a certain (symbolic) value, and 3) whether a vector of signals is equal to one of the few (symbolic) values. In these scenarios, our data abstraction is able to reduce the domain cardinality exponentially. Lastly, we will also demonstrate cases where our abstraction paves the way for other verification techniques such as bit-width reduction.

Memory abstraction Just like the data abstraction, our memory abstraction also aims at reducing the value field of the state space and utilizing the GV encoding's flexibility in encoding X to achieve that. However, it exclusively focuses on the memory signals and it simplifies the value transition function for a signal instead of the expression of its value field. The key goal of our memory abstraction is that in the value verification, irrelevant elements always have the same value as one of the care elements. To achieve this, it utilizes user-provided hints about what makes a memory element irrelevant to the verification to identify such elements and modifying their value transition functions. Therefore, though our memory abstraction does not actually reduces the number of memory elements, it achieves that in essence by making sure that in every cycle the irrelevant elements always have the same value as one of the care elements. To prove that ours is more powerful than the existing approach, we will show how the existing memory abstraction can be achieved with ours and how ours can be applied to memories such as CAM that the existing approach cannot handle. Lastly, we will provide templates of how to use it.

Refinement Recall that a ternary transition function abstracting a Boolean transition function is constructed by replacing the Boolean operators with the ternary ones. A key implication of this construction method is that the ternary transition function is dependent on the Boolean transition function in the sense that for two ternary transition functions constructed using different implementations of the same Boolean transition function, given the same ternary assignment to the inputs, they may return different ternary values that can be ordered by the abstraction relation defined by $0 \leq X, 1 \leq X$. For example, a multiplexer can be implemented by $o = (\text{sel} \wedge a) \vee (\neg \text{sel} \wedge b)$ or $o = (\text{sel} \wedge a) \vee (\neg \text{sel} \wedge b) \vee (a \wedge b)$. But given the ternary assignment ($a = 1, b = 1, \text{sel} = X$) the first ternary multiplexer returns X while the second one returns 1. Our refinement framework extends this idea and modifies the design in a way that increases the precision of its ternary abstraction. The refinement framework includes three approaches that trade off between completeness and ease of use. Fundamentally, our refinement methods and existing approaches that modify the ternary input stimulus and initial state all eliminate spurious counter-examples (CEXs) by partitioning them into more informative ones such that the property doesn't evaluate

to X . However, the key difference is that our approaches partition exactly at the parts of the traces where the information is lost while the existing approaches affect much more logic than needed. In other words, our refinement preserves the abstraction level better, which translates to better verification performance. Several of the designs we verify in the benchmarking require refinement, where we demonstrate the effectiveness of our refinement approaches.

Implementation and benchmarking We use symbolic indexing to construct a ternary functional vector which is a vector of ternary expressions to symbolically represent a set of ternary states. Symbolic simulation is used to compute reachable states of each cycle and AIG is used as the data structure for Boolean expressions. The SAT solver we use is the built-in Minisat [47] of ABC [11]. ABC is an academic open-source logic synthesis and verification suite. We demonstrate the data abstraction and memory abstraction by using them in bounded model checking (BMC) and k -induction. Our benchmarks include 6 varying designs and 8 properties and they differ in complexities, functionality and etc. We usually observe at least $10\times$ improvement over the state-of-art implementation of BMC in ABC (`bmc3`) due to our abstraction techniques especially as the verification problems are made more difficult by the increasing time bound and design parameters. We use off-the-shelf logic optimization algorithms to simplify the AIGs representing the verification problems. Our results show that the guard verification is simple enough such that it can be solved by the logic optimizations directly most of the time. The value verification is also consistently made simple enough by the logic optimizations such that the speedup in the SAT solving time is more than enough to cover the optimization time. These results demonstrate the great synergy between our data abstraction, memory abstraction and logic optimizations.

The rest of the thesis is organized as follows. Chapter 2 present the needed background knowledge and related work. Chapter 3 presents the guard-value encoding and compares it against the most widely used dual-rail encoding. This chapter explains what makes the guard-value encoding critical to our data abstraction (Chapter 5) and memory abstraction (Chapter 6). Chapter 4 analyzes what could make ternary verification better than the Boolean verification. It also compares ternary verification based on different encodings. The analysis and comparison are done using the bounded model checking as an example. Our refinement framework is presented in Chapter 7. Section 8.1 describes the designs and properties we use as benchmarks for our proposed techniques. The results are provided in Section 8.2. Lastly, Chapter 9 is the conclusion and future work.

Chapter 2

Background and Related Work

In this chapter, we provide the basis that our work builds on and describe related work. We first formalize the verification problems by describing our transition system that models hardware designs, and defining the satisfiability relation between a property and a transition system. As in done in STE, we represent a set of states as a Boolean or ternary functional vector, which is called symbolic indexing. For sets of Boolean states, this representation is also known as Boolean functional vector in the literature. We extend BMC and induction to ternary domain. Later, we present the abstraction relation between a set of Boolean states and a set of ternary states, which will be lifted to the abstraction relation between a Boolean transition system and a ternary transition system. This provides the foundation for the soundness of our verification approach. Next, we will discuss the implementation of ternary domain and ternary transition system, specifically we will talk about the encoding of the ternary domain, which in turn decides how to reduce the simulation/unrolling of a ternary transition system to simulating/unrolling two Boolean transition systems. Lastly, we conclude this chapter with related work.

2.1 Signals and States

A Boolean domain $\{0, 1\}$ is denoted by $\mathbb{B}$ and a ternary domain $\{0, 1, \mathbf{X}\}$ is denoted by $\mathbb{T}$. We will use S to denote the set of signals of the design. There are 4 types of signals of a hardware design, which are primary inputs, primary outputs, registers and combinational signals¹. The set of primary inputs is denoted as S_I while the set of the remaining signals is denoted as S_R . Signals may change values from cycle to cycle. A Boolean (ternary) state is a Boolean (ternary) assignment to signals in S , denoted as $q : S \mapsto \mathbb{B}$ ($\hat{q} : S \mapsto \mathbb{T}$). When there is no ambiguity, we may use q also to denote a ternary state. The state for primary inputs is denoted by q_I , while for the remaining signals, it is denoted by q_R . We

¹Combinational signals are the outputs of internal gates.

use $\mathcal{Q}(\hat{\mathcal{Q}})$ for a set of (ternary) states and $\mathbb{Q}(\hat{\mathbb{Q}})$ for the set of all possible (ternary) states, i.e., $\mathbb{Q} = \{q : S \mapsto \mathbb{B}\}$ ($\hat{\mathbb{Q}} = \{\hat{q} : S \mapsto \mathbb{T}\}$).

A set of states is usually characterized by a predicate in terms of the design signals. Alternatively, we can represent a set of Boolean (or ternary) states with a symbolic Boolean (or ternary) state. We define a symbolic Boolean state as a mapping from S to symbolic values (i.e., Boolean expressions), which also include 0 and 1. The idea of using a symbolic ternary state to represent a set of ternary states can be found in STE [46], referred to as symbolic indexing. In STE, a symbolic ternary state maps a signal from S to an expression of the form $(e_{guard} ? e_{value} : X)$, where e_{guard} and e_{value} can be any Boolean expressions. Since e_{guard} is called “guard” in STE, we will refer to such expression as guard expression hereafter. A guard expression maps an assignment to the Boolean variables to a ternary value. Note that a guard expression reduces to a Boolean expression if e_{guard} is 1. The expression $e_{g_1} ? 1 : (e_{g_0} ? 0 : X)$ is also a guard expression since it can be rewritten to $(e_{g_1} \vee e_{g_0}) ? e_{g_1} : X$.

The Boolean variables that appear in Boolean expressions and guard expressions are denoted as V . In contrary to signals from S , variables in V do not change their values from cycle to cycle. An assignment to variables in V is called an environment assignment denoted as $env : V \mapsto \mathbb{B}$. We will use a symbolic state to represent a set of states for the rest of the paper. Given the symbolic values for the primary inputs and the initial symbolic values of the registers, using symbolic simulation, we can compute the symbolic state in any clock cycle.

Example 2.1.1 and Example 2.3.1 show how to construct the symbolic state for inputs when they are unconstrained and constrained respectively. Given an environment env , a symbolic state evaluates to a concrete state, denoted as $\mathcal{Q}_{|env}$. Therefore, the set of concrete states represented by a symbolic state ($\mathcal{Q}$) is generated by evaluating $\mathcal{Q}$ over all environments, i.e., $\{\mathcal{Q}_{|env} : env \in (V \mapsto \mathbb{B})\}$. If we decide an order for the signals in S , states can be represented as vectors. A symbolic Boolean (ternary) state then becomes a vector of Boolean (ternary) expressions.

Example 2.1.1 (Unconstrained inputs). *Consider a design with 2 inputs that are unconstrained in every clock cycle. Because they are unconstrained, the set of states for the inputs includes every assignment to the inputs $\{(0, 0), (0, 1), (1, 0), (1, 1)\}$, which can be represented by the symbolic state (v_0, v_1) , where v_0 and v_1 are fresh Boolean variables.*

The representation we use here for a set of concrete Boolean states is also known as functional vector in the literature, for example [21, 27]. Parametric representation [1] is a technique to construct a symbolic Boolean state from a set of Boolean states characterized by a predicate.

2.2 Symbolic Indexing

Symbolic indexing (SI) represents multiple ternary states with a symbolic ternary state that maps each signal to a guard expression. In STE, it is used as part of STE’s specification language. The term, “symbolic indexing”, is perhaps due to one particular way to construct the symbolic ternary state. In this construction, a vector of Boolean variables, interpreted as a binary representation for a bounded integer, is used as an index into an array of ternary states. This construction method is illustrated in Example 2.2.1.

Example 2.2.1. *Consider the set of ternary states for design signals a , b and c :*

$$\{(1, X, X), (X, 1, X), (X, X, 1), (0, 0, 0)\}$$

Using symbolic indexing to represent the set as a symbolic state, we need 2 fresh Boolean variables, i_0 and i_1 , because there are 4 concrete states. We use (i_1, i_0) to select among the array of 4 states. For example, a is assigned 0 in the first assignment, assigned 1 in the 3rd assignment and assigned X for the rest. Therefore, the symbolic ternary state $\hat{Q}$ is:

$$\begin{aligned}\hat{Q}(a) &= \neg i_1 \wedge \neg i_0 ? 1 : (i_1 \wedge i_0 ? 0 : X) \\ \hat{Q}(b) &= \neg i_1 \wedge i_0 ? 1 : (i_1 \wedge i_0 ? 0 : X) \\ \hat{Q}(c) &= i_1 \wedge \neg i_0 ? 1 : (i_1 \wedge i_0 ? 0 : X)\end{aligned}$$

In fact, this method can be easily adapted to constructing a symbolic Boolean state to represent a set of Boolean states. However, note that to use this method, a set of ternary states has to be provided. [36] and [2] present a method for SI that avoids this requirement.

2.3 Transition System

A Boolean transition system ($\mathcal{M} = (\mathring{Q}_R, \text{Tr}, \Sigma_I)$) has three components:

- $\mathring{Q}_R$: the set of initial states
- $\text{Tr} : \mathbb{Q}_R \times \mathbb{Q}_I \mapsto \mathbb{Q}_R$: the transition function. In a hardware design, each gate’s output is the result of the Boolean function the gate represents with the gate’s inputs as arguments. A register’s output is its input’s value from the previous cycle.
- $\Sigma_I = \Sigma_I^0, \Sigma_I^1, \dots$: the set of stimulus, which is a set of sequences of input signals’ state. Σ_I^i is an input state at the i -th cycle.

Example 2.3.1 represents a set of sequences of input states with a sequence of symbolic states, i.e., symbolic stimulus.

Example 2.3.1. Consider a design where the input signals n_0 and n_1 have the following relation: n_1 cannot be 1 unless n_0 has been 1 in one of the previous cycles. In other words, n_0 and n_1 both are unconstrained in the current cycle if n_0 is assigned 1 in one of the previous cycles. Otherwise, n_0 is unconstrained but n_1 is forced to be 0. The set of satisfying concrete stimulus for n_0 and n_1 is:

- $(0, 0), (0, 0), \dots$
- $(0, 0), (1, 0), \dots$
- $(1, 0), (0, 0), \dots$
- $(1, 0), (0, 1), \dots$
- $(1, 0), (1, 0), \dots$
- $(1, 0), (1, 1), \dots$
- $\dots, \dots$

This set of concrete stimulus can be symbolically represented by: $(\Sigma_I^0, \Sigma_I^1, \dots)$, where

$$\begin{aligned}\Sigma_I^t(n_0) &= v_0^t, \\ \Sigma_I^t(n_1) &= \left(\bigvee_{j < t} v_0^j\right) ? v_1^t : 0.\end{aligned}$$

For any t , v_0^t and v_1^t are fresh Boolean variables from V .

Definition 1 (Traces, language). A sequence of concrete states (denoted by π) is a trace of $\mathcal{M} = (\mathring{Q}_R, Tr, \Sigma_I)$ iff $\pi_R^0 \in \mathring{Q}_R$, $\pi_I \in \Sigma_I$ and $\pi_R^t = Tr(\pi_R^{t-1}, \pi_I^{t-1})$ for any cycle t that is not longer than the length of the sequence. Traces of $\mathcal{M}$ are denoted by Σ . The language of $\mathcal{M}$, denoted by $\mathcal{L}(\mathcal{M})$ is the set of all traces of $\mathcal{M}$.

For example, consider a timer (t) that starts from 0 and the input toggles (in) whether to pause the timer. A trace of length 3 without pause is $(in = 0, t = 0), (in = 0, t = 1), (in = 0, t = 2)$. Traces in the language of the timer have the first three states as:

- $(in = 0, t = 0), (in = 0, t = 1), (in = 0, t = 2)$
- $(in = 1, t = 0), (in = 0, t = 0), (in = 0, t = 0)$
- $(in = 1, t = 0), (in = 1, t = 0), (in = 0, t = 1)$
- $(in = 1, t = 0), (in = 1, t = 0), (in = 1, t = 1)$
- $(in = 0, t = 0), (in = 1, t = 1), (in = 0, t = 1)$

- $(\text{in} = 0, \text{t} = 0), (\text{in} = 1, \text{t} = 1), (\text{in} = 1, \text{t} = 1)$
- $(\text{in} = 0, \text{t} = 0), (\text{in} = 0, \text{t} = 1), (\text{in} = 0, \text{t} = 2)$
- $(\text{in} = 0, \text{t} = 0), (\text{in} = 0, \text{t} = 1), (\text{in} = 1, \text{t} = 2)$

For simplicity, we use $\text{Tr}^t(\mathring{\mathcal{Q}}_R, \Sigma_I)$ to denote the set of reachable states for signals in R in cycle t , which can be iteratively computed:

$$\begin{aligned} \text{Tr}(\mathring{\mathcal{Q}}_R, \Sigma_I) &= \text{Tr}(\mathring{\mathcal{Q}}_R, \Sigma_I^0) \\ \text{Tr}^t(\mathring{\mathcal{Q}}_R, \Sigma_I) &= \text{Tr}(\text{Tr}^{t-1}(\mathring{\mathcal{Q}}_R, \Sigma_I), \Sigma_I^{t-1}) \text{ for any } t \geq 2 \end{aligned} \quad (2.1)$$

This iterative computation is also called symbolic simulation. We sometimes refer to Tr^t as unrolled transition function. If $\mathring{\mathcal{Q}}_R$ and Σ_I are represented symbolically, the language of the design can be generated by the following symbolic trace:

$$\left(\mathring{\mathcal{Q}}_R, \quad \text{Tr}(\mathring{\mathcal{Q}}_R, \Sigma_I), \quad \text{Tr}^2(\mathring{\mathcal{Q}}_R, \Sigma_I), \quad \text{Tr}^3(\mathring{\mathcal{Q}}_R, \Sigma_I), \quad \dots \right) \quad (2.2)$$

It is straightforward to extend the definition of Boolean transition system and traces to the ternary domain, denoted by $\hat{\mathcal{M}} = (\hat{\mathcal{Q}}, \hat{\text{Tr}}, \hat{\Sigma}_I)$, and $\hat{\pi}$ respectively.

If Tr is represented using an And-Inverter-Graph (AIG). The ternary transition function $\hat{\text{Tr}}$ can be derived from Tr by replacing the Boolean AND gate and NOT gate with the corresponding ternary ones. The semantics of the ternary gates are described in Table 2.1.

Table 2.1: Ternary AND ($\hat{\wedge}$) and NOT ($\hat{\neg}$)

$\hat{\wedge}$	0	1	X
0	0	0	0
1	0	1	X
X	0	X	X

$\hat{\neg}$	0	1	X
0	1	0	X
1	0	1	X
X	X	X	X

2.4 BMC and Inductive Proof in Ternary Domain

Let ϕ be a predicate over signals ($\phi: \mathbb{Q} \mapsto \mathbb{B}$). A predicate maps an assignment to the signals to a Boolean value. We verify that for every clock cycle ϕ should be 1 for the design under verification, i.e., ϕ is an invariant of the design. We discuss two common techniques to verify such safety properties, which are BMC and induction. BMC is commonly used as a bug hunting technique because the diameter of a practical design is usually formidable for BMC. Induction is the simplest yet very useful proof technique. For each technique, we will first describe how to verify a Boolean transition system. Then we will extend it to ternary transition system.

2.4.1 Bounded Model Checking

In order to use BMC to check the correctness in cycle t , we need to compute the set of reachable states for the signals in R in cycle t , denoted by Σ_R^t . Recall that as shown in (2.2), it is computed by symbolically simulating the design for t clock cycles starting from the symbolic initial state $\hat{Q}_R$ and using the input stimulus Σ_I .

When there are two functions with different domains, such as $f : A \mapsto C$ and $g : B \mapsto C$, then we use $f \cup g : A \cup B \mapsto C$ to denote:

$$(f \cup g)(n) = \begin{cases} f(n) & \text{if } n \in A \\ g(n) & \text{if } n \in B \end{cases}$$

Definition 2. $\mathcal{M}$ satisfies ϕ in cycle t , denoted by $\mathcal{M} \models^t \phi$, iff the formula $\neg\phi(\Sigma_R^t \cup \Sigma_I^t)$ is unsatisfiable.

Computing the reachable *ternary* states in the cycle t is the same process:

$$\hat{\Sigma}_R^t = \hat{\text{Tr}}^t(\hat{Q}_R, \hat{\Sigma}_I)$$

Naturally, we extend the domain of a predicate ϕ to accommodate the ternary assignments (Definition 3).

Definition 3 (Ternary predicate). *A ternary predicate maps a ternary assignment to a value from ternary domain: $\hat{\phi} : \hat{\mathbb{Q}} \mapsto \mathbb{T}$. For any predicate ϕ , its corresponding ternary predicate $\hat{\phi}$ is constructed by interpreting the logic operators inside ϕ with ternary logic. AND and NOT make up a complete set of logic operators. Their interpretation in ternary logic has been given in Table 2.1.*

Note that over a ternary assignment, a ternary predicate can be evaluated to $\mathbf{X}$, which indicates the assignment lacks information to decide whether it satisfies the predicate. In other words, there are two different Boolean assignments from the ternary assignment's MRS, such that the predicate evaluates to different values. Definition 4 defines ternary bounded satisfiability. Note that here we explicitly compares the ternary predicate with 1 because it could evaluate to $\mathbf{X}$.

Definition 4. $\mathcal{M} \models^t \hat{\phi}$ iff $\hat{\phi}(\hat{\Sigma}_R^t \cup \hat{\Sigma}_I^t) = 1$ is unsatisfiable.

2.4.2 Induction

Induction works in two steps: the base step and the induction step. In the base step, it checks whether there is a counter-example (CEX) of length 1 from an initial state from $\hat{Q}_R$, which is BMC at bound 1 for $\mathcal{M} = (\hat{Q}_R, \text{Tr}, \Sigma_I)$. In the induction step, we check

whether for any state that satisfies ϕ , its next state also satisfies ϕ . Effectively, it amounts to bounded model checking the modified transition system $\mathcal{M}' = (\mathring{Q}_{free}, \text{Tr}, \Sigma_I)$ at bound 1 under the condition that its initial states satisfy ϕ (i.e., Formula 2.3). $\mathring{Q}_{free}$ is a symbolic state that maps all signals from S_R to fresh Boolean variables.

$$\phi(\mathring{Q}_{free} \cup \Sigma_I^0) \implies \phi(\text{Tr}(\mathring{Q}_{free}, \Sigma_I^0) \wedge \Sigma_I^1) \quad (2.3)$$

If Formula 2.3 is unsatisfiable, ϕ is inductive. If ϕ is inductive and the base step passes, we have a proof that the design always satisfies ϕ .

As discussed, an induction proof is reduced to two BMCs on the original transition system and the transition system with modified initial state respectively. As a result, it is straightforward to extend induction to the ternary domain. In the base step, we check satisfiability of

$$\hat{\phi}(\mathring{Q}_R \cup \hat{\Sigma}_I^0) = 1$$

In the induction step, we check the satisfiability of

$$\left(\hat{\phi}(\mathring{Q}_{free} \cup \hat{\Sigma}_I^0) = 1 \right) \implies \left(\hat{\phi}(\text{Tr}(\mathring{Q}_{free}, \hat{\Sigma}_I^0) \cup \hat{\Sigma}_I^1) = 1 \right)$$

Note that here in the modified ternary transition system, its initial state also maps state variables to fresh Boolean variables. Therefore, we keep using the notation $\mathring{Q}_{free}$.

k -induction can be reduced to multiple BMCs. Extending k -induction to ternary transition systems can be done similarly. IC3 [9] has many queries that check whether a formula is inductive and safe. It could be interesting to replace it with ternary induction. However, it will not be straightforward because one ternary abstraction (to be described in the next section) may not be accurate enough for every query of inductive checks.

2.5 Ternary Abstraction

We define the abstraction relation as a partial order and denote it with $\leq$. Among the values from the ternary domain, $\mathbf{X}$ abstracts both of 0 and 1 because it has the least amount of information (Definition 5). A signal with the value $\mathbf{X}$ implies that we do not know whether it has value 0 or 1.

Definition 5. 1) $0 \leq \mathbf{X}$, 2) $1 \leq \mathbf{X}$, 3) $0 \leq 0$, $1 \leq 1$, $\mathbf{X} \leq \mathbf{X}$

In the following definition, we define operators $\sqcup$ which computes the value that is the least upper bound of the operands: the least abstracted one among the values that abstract both operands. We also define $\sqcap$ which is the greatest lower bound of the operands: the most abstracted one among the values that are abstracted by both operands.

Definition 6.

- $0 \sqcup 0=0, 0 \sqcup 1=X, 0 \sqcup X=X$
- $1 \sqcup 1=1, 1 \sqcup X=X, X \sqcup X=X$
- $0 \sqcap 0=0, 0 \sqcap 1= \perp, 0 \sqcap X=0$
- $1 \sqcap 1=1, 1 \sqcap X=1, X \sqcap X=X$

$\perp$ is a special value used to indicate the there is no value that is abstracted by both operands.

Next, we extend this definition to states. A state is an abstraction of another state, iff for any signal, it is assigned a more abstracted value in the abstraction state (Definition 7).

Definition 7. $q \leq q'$ iff $\forall n \in S . q(n) \leq q'(n)$.

A ternary state could abstract multiple Boolean states. Let us call the set of all Boolean states that are abstracted by a ternary state “maximum representable set” (MRS) of the ternary state. A ternary state has a unique MRS and a MRS corresponds to a unique ternary state. For example, $\{(0, 0, 1), (1, 1, 1)\}$ and $\{(0, 1, 1), (1, 0, 1)\}$ are both abstracted by $(X, X, 1)$. The MRS for $(X, X, 1)$ is $\{(0, 0, 1), (0, 1, 1), (1, 0, 1), (1, 1, 1)\}$.

Definition 8.

- $q \sqcup q' = \lambda n : q(n) \sqcup q'(n)$
- $q \sqcap q' = \lambda n : q(n) \sqcap q'(n)$.
- $q \sqcap q' = \perp$ iff there exists a signal n such that $q(n) \sqcap q'(n) = \perp$.

A set of states is an abstraction of another set of states iff for all states in the other set, there exists one state that is more abstracted from the abstraction set of the states (Definition 9).

Definition 9. $\mathcal{Q} \leq \mathcal{Q}'$ iff $\forall q \in \mathcal{Q} . \exists q' \in \mathcal{Q}' . q \leq q'$

For a set of ternary states, its MRS is defined as the union of the MRSs for each ternary state. As shown in the following example, different ternary sets may share the same maximum representable set.

Example 2.5.1.

- $\{(0, 1, 1), (0, 0, 1)\} \leq \{(X, X, 1)\}$ because $(0, 1, 1) \leq (X, X, 1)$ and $(0, 0, 1) \leq (X, X, 1)$.
- $\{(0, X, 1), (1, X, 1)\} \leq \{(X, X, 1)\}$ because $(0, X, 1) \leq (X, X, 1)$ and $(1, X, 1) \leq (X, X, 1)$.
 $\{(0, X, 1), (1, X, 1)\}$ and $\{(X, X, 1)\}$ share the same maximum representable set, that is $\{(0, 0, 1), (0, 1, 1), (1, 0, 1), (1, 1, 1)\}$.

In Definition 3, we define ternary functions. Let $\hat{f}$ be a ternary function and f be the Boolean function that interpret operators in $\hat{f}$ using Boolean logic. The domains for $\hat{f}$ and f are denoted by $\mathbb{D}$ and $\hat{\mathbb{D}}$ respectively. Ternary functions have the following properties.

Proposition 1 (Properties of ternary functions).

- $\hat{f}$ agrees with f on Boolean inputs: $\hat{f}(y) = f(y)$, if $y \in \mathbb{D}$
- $\hat{f}$ is monotone with regard to the abstraction relation: $\hat{y} \leq \hat{y}' \implies \hat{f}(\hat{y}) \leq \hat{f}(\hat{y}')$.

For every signal $\mathbf{n} \in S_R$, its ternary transition function denoted by $\hat{\text{Tr}}(\mathbf{n})$ is a ternary function. Thus, it has the monotonicity:

$$\mathcal{Q}_R \leq \mathcal{Q}'_R \wedge \mathcal{Q}_I \leq \mathcal{Q}'_I \implies \hat{\text{Tr}}(\mathbf{n})(\mathcal{Q}_R, \mathcal{Q}_I) \leq \hat{\text{Tr}}(\mathbf{n})(\mathcal{Q}'_R, \mathcal{Q}'_I)$$

A ternary transition system is an abstraction of another Boolean or ternary transition system iff for any trace of the original transition systems, there is an abstracted trace in the abstraction transition system (Definition 10).

Definition 10. $\mathcal{M} \leq \hat{\mathcal{M}}'$ iff $\forall \sigma_R \in \mathcal{L}(\mathcal{M}) . \exists \hat{\sigma}'_R \in \mathcal{L}(\hat{\mathcal{M}}') . \sigma_R \leq \hat{\sigma}'_R$. Similarly, between two ternary transition systems, $\hat{\mathcal{M}} \leq \hat{\mathcal{M}}'$ iff $\forall \hat{\sigma}_R \in \mathcal{L}(\hat{\mathcal{M}}) . \exists \hat{\sigma}'_R \in \mathcal{L}(\hat{\mathcal{M}}') . \hat{\sigma}_R \leq \hat{\sigma}'_R$.

Lemma 1 states that abstraction is achieved by manipulating the initial states and the stimulus. Lemma 2 shows that our ternary transition function is just as precise as the original Boolean transition function. Furthermore, it shows that the abstracted transition systems are ordered by the initial states and the stimulus. In Example 2.5.2, you can find concrete examples of a Boolean transition system, a ternary transition system that abstracts it and a ternary transition system that is equivalent to it.

Lemma 1. $\hat{\mathcal{M}} = (\hat{\mathcal{Q}}_R, \hat{\text{Tr}}, \hat{\Sigma}_I)$ is an abstraction of $\mathcal{M} = (\mathcal{Q}_R, \text{Tr}, \Sigma_I)$ iff $\mathcal{Q}_R \leq \hat{\mathcal{Q}}_R$ and $\Sigma_I \leq \hat{\Sigma}_I$. Similarly, $\hat{\mathcal{M}}' = (\hat{\mathcal{Q}}'_R, \hat{\text{Tr}}, \hat{\Sigma}'_I)$ is an abstraction of $\hat{\mathcal{M}} = (\hat{\mathcal{Q}}_R, \hat{\text{Tr}}, \hat{\Sigma}_I)$ iff $\hat{\mathcal{Q}}_R \leq \hat{\mathcal{Q}}'_R$ and $\hat{\Sigma}_I \leq \hat{\Sigma}'_I$.

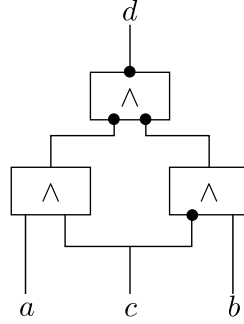
Lemma 2. $\hat{\mathcal{M}}_{\perp} = (\hat{\mathcal{Q}}_R, \hat{\text{Tr}}, \hat{\Sigma}_I)$ is equivalent to $\mathcal{M} = (\mathcal{Q}_R, \text{Tr}, \Sigma_I)$.

Example 2.5.2 (Abstraction with ternary transition system). Let us consider $\mathcal{M} = (\mathcal{Q}_R, \text{Tr}, \Sigma_I)$, where the transition function is $\text{Tr}(\mathbf{a}, \mathbf{b}, \mathbf{c}) = \mathbf{c} ? \mathbf{a} : \mathbf{b}$. Figure 2.1 shows an AIG implementation of Tr , where the solid dot represents a NOT gate. Assume that, for each cycle, the input state maps $\mathbf{a}$ and $\mathbf{b}$ to 1, and $\mathbf{c}$ to a fresh Boolean variable v . After symbolic simulation, $\mathcal{Q}_R(\mathbf{d}) = 1$ for any cycle t .

The first ternary transition system we consider is $\hat{\mathcal{M}} = (\hat{\mathcal{Q}}_R, \hat{\text{Tr}}, \hat{\Sigma}_I)$, where $\hat{\Sigma}_I$ maps $\mathbf{a}$ and $\mathbf{b}$ to 1 and $\mathbf{c}$ to X for each cycle. As a result, $\Sigma_I \leq \hat{\Sigma}_I$ and $\mathcal{M} \leq \hat{\mathcal{M}}$. After ternary symbolic simulation, $\hat{\mathcal{Q}}_R(\mathbf{d}) = \text{X}$ for any cycle t , which indeed is an abstraction of $\mathcal{Q}_R$.

Lastly, let us consider the most accurate ternary transition system from Lemma 2, where we use the same stimulus as from $\mathcal{M}$. After ternary symbolic simulation, $\hat{\mathcal{Q}}_R(\mathbf{d}) = 1$ for any cycle t . This demonstrates that the abstraction level of the ternary transition system solely depends on $\hat{\mathcal{Q}}$ and $\hat{\Sigma}_I$, instead of $\hat{\text{Tr}}$.

Figure 2.1: An AIG representation of a multiplexer



Theorem 1 is the basis for the correctness of our approach. Its proof directly follows the definition of transition system abstraction in Definition 10.

Theorem 1. *Let $\hat{\mathcal{M}}$ be a ternary abstraction of $\mathcal{M}$.*

1. *if there is no CEX within bound k of $\hat{\mathcal{M}}$, there is no CEX within bound k of $\mathcal{M}$, i.e., $\hat{\mathcal{M}} \models^k \hat{\phi} \implies \mathcal{M} \models^k \phi$*
2. *if $\hat{\mathcal{M}}$ is inductive, $\mathcal{M}$ is inductive*

Proof. To prove 1, let's prove that if $\mathcal{M} \not\models \phi$ then $\hat{\mathcal{M}} \not\models \hat{\phi}$. Let there be a trace σ from $\mathcal{L}(\mathcal{M})$ such that it fails ϕ in the cycle k , i.e., $\phi(\sigma^k) = 0$. According to Definition 10 and because of $\mathcal{M} \leq \hat{\mathcal{M}}$, there exists $\hat{\sigma}$ that abstracts σ : $\sigma \leq \hat{\sigma}$. Because $\hat{\phi}$ is monotone in terms of the abstraction relation and $\hat{\phi}(\sigma) = 0$, we have $\hat{\phi}(\hat{\sigma}) = 0$ or $\hat{\phi}(\hat{\sigma}) = \mathbf{X}$, i.e., $\hat{\mathcal{M}} \not\models \hat{\phi}$.

The proof of 2 will be similar. □

2.6 Ternary Encoding

Recall that transition functions and predicates are represented in AIGs. We will show how to represent a ternary AND and NOT gate with Boolean gates. This way, we reduce a ternary symbolic simulation to two Boolean symbolic simulations.

First, let us use a pair of Boolean values to encode a ternary value from $\mathbb{T}$. The most widely used encoding is called dual-rail encoding (Definition 11), where the left of the pair is called “high rail” and the right one is called “low rail”. Intuitively, the high rail indicates whether the ternary value could be 1 and the low rail indicates whether the ternary value could be 0. $\mathbf{X}$ could be either 1 or 0. As a result, it is represented by $\langle 1, 1 \rangle$.

Definition 11. *In dual-rail encoding, $\langle 1, 0 \rangle$ represents 1. $\langle 0, 1 \rangle$ represents 0 and $\langle 1, 1 \rangle$ represents $\mathbf{X}$.*

A symbolic ternary value can be represented by a pair of Boolean functions. Thus, a symbolic ternary state can be represented by a vector of pairs of Boolean functions. However, for the ease of reading, we prefer to use two vectors of Boolean functions for the high rails and low rails. We will refer to them high rail state/vector and low rail state/vector. There is a straightforward translation between the notations. We illustrate it here. Let there be k state variables and the symbolic state be $(\langle h_0, l_0 \rangle, \langle h_1, l_1 \rangle, \dots, \langle h_{k-1}, l_{k-1} \rangle)$, where $h_0, h_1, \dots, h_{k-1}$ and $l_0, l_1, \dots, l_{k-1}$ are Boolean functions. Then, the two vectors we prefer to use are $(h_0, h_1, \dots, h_{k-1})$ and $(l_0, l_1, \dots, l_{k-1})$. We now show a concrete example of using high rail state and low rail state to represent a ternary symbolic state.

Example 2.6.1. *Let us consider the symbolic state for signals a , b and c from Example 2.2.1, which is*

$$\begin{aligned} &(\neg i_1 \wedge \neg i_0 ? 1 : (i_1 \wedge i_0 ? 0 : X) , \\ &\neg i_1 \wedge i_0 ? 1 : (i_1 \wedge i_0 ? 0 : X) , \\ &i_1 \wedge \neg i_0 ? 1 : (i_1 \wedge i_0 ? 0 : X)) \end{aligned}$$

The low rail state is $(\neg(\neg i_1 \wedge \neg i_0), \neg(\neg i_1 \wedge i_0), \neg(i_1 \wedge \neg i_0))$ because signals a , b and c are 1 iff $\neg i_1 \wedge \neg i_0$, $\neg i_1 \wedge i_0$ and $\neg(i_1 \wedge \neg i_0)$ are 1 respectively. Similarly, the high rail state is $(\neg(i_1 \wedge i_0), \neg(i_1 \wedge i_0), \neg(i_1 \wedge i_0))$.

Based on the Table 2.1, the truth tables for the high rail and low rail of the output of ternary AND is shown in Table 2.2. Table 2.3 shows the two rails of the output of a ternary NOT gate.

Table 2.2: The high rail and low rail of the output of a ternary AND ($\hat{\wedge}$)

$\hat{\wedge}$ $\langle \text{high rail}, \text{low rail} \rangle$	$\langle 0, 1 \rangle$	$\langle 1, 0 \rangle$	$\langle 1, 1 \rangle$
$\langle 0, 1 \rangle$	$\langle 0, 1 \rangle$	$\langle 0, 1 \rangle$	$\langle 0, 1 \rangle$
$\langle 1, 0 \rangle$	$\langle 0, 1 \rangle$	$\langle 1, 0 \rangle$	$\langle 1, 1 \rangle$
$\langle 1, 1 \rangle$	$\langle 0, 1 \rangle$	$\langle 1, 1 \rangle$	$\langle 1, 1 \rangle$

Table 2.3: The high rail and low rail of the output of a ternary NOT ($\hat{\neg}$)

$\hat{\neg}$ $\langle \text{high rail}, \text{low rail} \rangle$	$\langle 0, 1 \rangle$	$\langle 1, 0 \rangle$	$\langle 1, 1 \rangle$
	$\langle 1, 0 \rangle$	$\langle 0, 1 \rangle$	$\langle 1, 1 \rangle$

Let the inputs to a ternary AND have values $\langle a_h, a_l \rangle$ and $\langle b_h, b_l \rangle$. Then based on Table 2.2, the output's value, denoted as $\langle o_h, o_l \rangle$ are:

$$\begin{aligned} o_h &= a_h \wedge b_h \\ o_l &= a_l \vee b_l \end{aligned} \tag{2.4}$$

Similarly, the output's value of a ternary NOT gate can be computed in terms of its input's value, denoted as $\langle \mathbf{a}_h, \mathbf{a}_l \rangle$:

$$\begin{aligned} \mathbf{o}_h &= \mathbf{a}_l \\ \mathbf{o}_l &= \mathbf{a}_h \end{aligned} \tag{2.5}$$

According to Equation 2.4 and Equation 2.5, symbolically simulating a ternary AND/NOT gate is reduced to Boolean symbolic simulation to compute the high-rail and the low-rail of the output. Notably, the output of the ternary NOT swaps the rails of its input value. As a result, for both rails of any signal from S_R , its current value will depend on both rails of the registers' values in the previous cycle and both rails of the primary inputs' values in the current cycle. We now show how to construct a Boolean transition system $\mathcal{M}_{dr}$ which can be used to compute both rails of a signal's value, from $\hat{\mathcal{M}} = (\hat{\mathcal{Q}}_R, \hat{\text{Tr}}, \hat{\Sigma}_I)$.

Firstly, $\mathcal{M}_{dr}$ doubles the number of signals in $\hat{\mathcal{M}}$ to incorporate both rails of the signals: for each signal $\mathbf{n} \in S$, $\mathbf{n}_h$ and $\mathbf{n}_l$ are the signals in $\mathcal{M}_{dr}$. Let $\hat{\mathcal{Q}}_{R_h} : S_R \mapsto \mathbb{B}$ and $\hat{\mathcal{Q}}_{R_l} : S_R \mapsto \mathbb{B}$ be the high rail and low rail component of $\hat{\mathcal{Q}}_R$ respectively, i.e.,

$$\forall \mathbf{n} \in S_R . (\hat{\mathcal{Q}}_{R_h}(\mathbf{n}), \hat{\mathcal{Q}}_{R_l}(\mathbf{n})) = \hat{\mathcal{Q}}_R(\mathbf{n})$$

Similarly, Σ_{I_h} and Σ_{I_l} are the high rail and low rail components of $\hat{\Sigma}_I$. Recall that we use 'U' to combine functions with distinct domains. $\mathcal{M}_{dr} = (\hat{\mathcal{Q}}_{R_h} \cup \hat{\mathcal{Q}}_{R_l}, \text{Tr}_{dr}, \Sigma_{I_h} \cup \Sigma_{I_l})$ is the Boolean transition system where for each signal n of $\hat{\mathcal{M}}$, $\mathbf{n}_h$ and $\mathbf{n}_l$ are mapped to the high rail and low rail of $\hat{q}_R(\mathbf{n})$, i.e., $\langle q_{dr}(\mathbf{n}_h), q_{dr}(\mathbf{n}_l) \rangle = \hat{q}_R(\mathbf{n})$. Tr_{dr} is constructed from $\hat{\text{Tr}}$ by replacing its ternary AND and NOT gates with their Boolean representations, which are shown in Formula 2.4 and Formula 2.5 if the encoding is dual rail encoding. We provide a concrete example of Tr_{dr} .

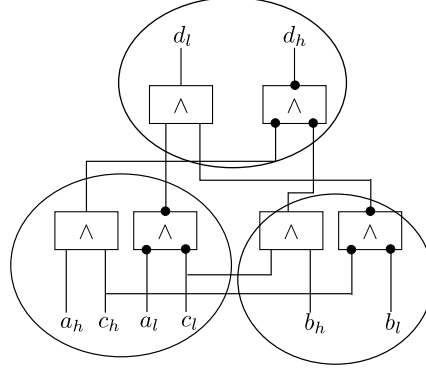
Example 2.6.2 (Boolean representation of a ternary multiplexer using dual-rail encoding). *Figure 2.1 is a multiplexer's AIG. Figure 2.2 shows the result after we substitute the ternary AND and NOT in the AIG with the Boolean gates based on Formula 2.4 and Formula 2.5. In each circle in Figure 2.2, there are two AND gates that generate the high rail and low rail of the original ternary AND.*

Using dual-rail encoding, for any signal $\mathbf{n} \in S_R$, the number of AND gates in its transition function $\text{Tr}_{dr}(\mathbf{n}_h)$ is the same as $\text{Tr}_{dr}(\mathbf{n}_l)$ and it is equal to the number of ternary AND gates in $\hat{\text{Tr}}(\mathbf{n})$. Therefore, as shown in Equation 2.6, for any cycle t and any signal $\mathbf{n} \in S$, the number of AND to represent both rails are the same and in total, there are twice as many as AND as in $\Sigma_R^t(\mathbf{n})$ (Lemma 3). Recall that Σ_R^t is the state of the original Boolean transition system at cycle t . In the next section, we will present a new ternary encoding and show its advantages over the dual-rail encoding.

Lemma 3.

$$|\Sigma_{dr}^t(\mathbf{n}_h)| = |\Sigma_{dr}^t(\mathbf{n}_l)| = |\Sigma_R^t(\mathbf{n})|, \text{ where } |\cdot| \text{ is the number of AND in a Boolean expression.} \tag{2.6}$$

Figure 2.2: An AIG representation of a ternary multiplexer using dual-rail encoding



2.7 Related Work

The guard-value encoding was first used by Chakraborty et al. in [13] where a word-level STE was proposed. However, it only focused on its advantages of compactly representing the guard of a vector with just 1 bit if all bits of the vector are either $\mathbf{X}$ or none- $\mathbf{X}$ at the same time.

There is a body of work on symbolic simulation and using symbolic simulation with model checking. Function vector representation goes hand in hand with symbolic simulation because for symbolic simulation to compute the next-state value for a signal n , it needs to know the current values of the signals in n 's support, which are conveniently stored as elements in the function vector representing the current state. Characteristic function is the dominant representation for lots of model checking algorithms, e.g., symbolic model checking [12], Interpolation Based Model Checking [35], PDR/IC3 [9][22]. Still there are several papers which worked on using functional vector representation instead. It is well known that BDD representation for functional vector can be exponentially more compact than the characteristic function representation [1]. Algorithms for applying set operations on Boolean functional vectors were given in [27], which enabled the integration of symbolic simulation into reachability analysis. Goel et al. [28] extends Boolean functional vector with partial order and showed that partially ordered BFVs can serve as abstractions for bit vector sets and can be used to compute over-approximations in reachability analysis. [15] gave a symbolic simulation based BMC that outperformed plain BMC. The key was to apply their efficient SAT-based reparameterization to the BFV when it becomes too complex during simulation. Later, [14] showed results of replacing plain BMC with symbolic simulation based BMC to compute abstract circuit model. And-Inverter-Graph (AIG) is a non-canonical representation, but the size of BFV represented in AIG increases linearly as clock cycles increase. Besides, nodes hashing and graph reduction methods can further reduce the size of the AIG, [26] showed the efficiency of BMC implemented with symbolic simulation where the Boolean expressions are represented in AIG and satisfiability is checked with SAT solver. State of art BMC implementations use AIG as intermediate representation before getting translated to CNF, because it has been shown that circuit-

based simplifications, such as SAT sweeping [32] [41], rewriting [7] [40], cut-based CNF translation from AIG [23], can significantly improve BMC performance. Constraints indicate care-set on the inputs. They can be combined with SAT-sweep to help simplify problem AIG [33]. In this thesis, we use ternary function vector as representations for sets of ternary states and ternary symbolic simulation to compute the reachable states in each cycle. A signal’s value is represented using AIG. The logic optimizations are able to simplify the verification problem because of the data abstraction and memory abstraction.

The existing data abstraction methods most closely related to our work are predicate abstraction [29] and Symbolic Trajectory Evaluation (STE) [46]. STE achieves data abstraction by representing a set of Boolean states with a smaller set of ternary states. Our data abstraction improves on STE and leverages the guard-value encoding’s flexibility in encoding X to further reduce the state space in value verification. STE has its own property specification language that has restrictive syntax that makes it less friendly than other property specification languages. In our presentation of data abstraction, the safety property to verify is implemented as a monitor circuit and combined with the design, which makes our data abstraction independent of the property specification language. In our data abstraction, logic optimizations play an important role which has not been discussed in STE. Predicate abstraction uses a set of predicates to partition the entire state space: an assignment to the set of predicates represents an abstract state and different concrete states that evaluate the predicates to the same value are mapped to the same abstract state. On the other hand, in our data abstraction, the predicates we use only partition the state space for a small number of selected signals while the other signals’ state space remains the same. This way, two different concrete states that evaluate the predicates to the same value but assigns different values to the signals, whose state space is not meant to be abstracted by the predicates, are mapped to two different abstract states. It is easier to come up with a set of predicates that captures enough information about the states to avoid spurious CEX for a subset of signals than for all of the signals. We believe it makes our data abstraction easier and more flexible to use than the predicate abstraction. In predicate abstraction, for each cycle, to compute the reachable abstract states, predicate abstraction has to solve satisfiability problems. In our data abstraction, because it relies on ternary logic to achieve abstraction, the set of values of the set of signals characterized by an assignments to the predicates has to be approximated by a set of ternary assignments. Each ternary assignment has to be accurate enough such that the predicates evaluate to a non- X value. In our data abstraction, computing the next cycle reachable states is just as efficient as in the concrete system and it is purely syntactically. But our data abstraction cannot handle infinite state systems like predicate abstraction.

In our data abstraction, to abstract the input state space or initial state space, users need to manually construct the set of ternary states abstracting the original state space. We provide guidelines for the construction and also present some exemplary data abstraction that can be adapted for a range of verification problems. Related work was done by Adams et al. in [2] and Melham et al. in [36]. Melham et al. in [36] defined the notion of indexing relations and given an indexing relation, authors proposed a method using strong preimage

to compute a set of ternary states abstracting a set of Boolean states. Adams et al. [2] developed an algorithm to compute an indexing relation automatically. Their automatic approach solely aims at reducing the number of ternary states which means fewer fresh variables are needed in the symbolic representation. However, our data abstraction stresses the importance of cardinality of the set of values states instead of the ternary states and also cares about the size of the AIG representation, which is not a factor considered in their automatic approach.

Similar to our data abstraction, symmetry reduction ([19, 25, 31]) also reduces the state space. But there are key differences. In symmetry reduction, two states are considered “equivalent” (belonging to the same class) if there exists a graph automorphism in the system’s transition graph and the bijection mapping associated with the automorphism maps these two states to one another. On the other hand, in our data abstraction, two Boolean states are considered “equivalent” if they are represented by the same ternary state. The equivalent states are not differentiated in the verification. For symmetry reduction, it is the quotient system, where equivalent states correspond to 1 state, that is verified. For our data abstraction, equivalent states have the same value field in their ternary representation. Therefore, they are essentially 1 state in the value verification. The guard verification can be regarded as checking that our state space reduction in the value field is correct. The other differences include that our techniques are based on symbolic simulation and utilize logic optimization algorithms to make guard and value verification efficient. The efficiency of our abstraction also requires that the correctness of a property can be defined with a small set of symbolic variables, i.e., the property uses (or can be rewritten into a property that uses) symbolic variables. On the other hand, symmetry reduction is a property of the system’s transition graph.

Memory abstraction [6] can be very effective for verifying memory or data intensive hardware designs. It replaces a full memory with only a few memory slots when they are all we need to reason the correctness of the property to verify. For example, when we verify that the read data is the last written data to the read address for a memory implementation, we only need one slot, whose address matches the read address. In details, the method proposed [6] traverses a word-level network to find memory that can be abstracted, which is then reduced to only a few slots that are addressed by symbolic constants. If a read operation whose address is not equal to one of the symbolic constants, the read data is arbitrary. Otherwise, the read data is what is stored in the slot whose address matches the read address. Our memory abstraction does not reduce the number of memory slots directly, instead it modifies each memory slot’s value transition function such that the irrelevant memory slots share the same value transition function, which essentially reducing the number of memory slots for value verification. Furthermore, our memory abstraction is more powerful than [6] because it can apply to memory such as CAM’s memory that cannot be done using the existing method.

Compositional model checking ([20, 34]), also known as assume-guarantee, decomposes the property to easier-to-verify local properties. Our data abstraction and memory abstraction are orthogonal to compositional model checking.

There are several papers about refinement in STE. Roorda et al. [44] presented methods to compute the weakest satisfying trace or unsatisfying trace of an STE property, which can aid users to refine STE properties. Automatic refinement approaches were presented by Tzoref et al. [48] and Chockler [17]. When encountering a spurious CEX, they both pick the signal that is in the consequent, contains X in the CEX, and has the least number of inputs in its support. In order to eliminate the X on the target signal, both papers pick the inputs that are assigned X in the CEX and are the likely cause of X on the target signal. The selected inputs will be assigned fresh symbolic variables in the refined STE property. [48] uses a heuristic that prefers the inputs affecting control to those affecting data. [17], instead, computes a metric called ‘degree of responsibility’ (DoR) for each input. Then the inputs with the highest degree of responsibility are picked. Degree of responsibility for a signal is the inversion of how many other inputs have to change values in order for X to occur on that signal. The exact DoR is expensive to compute. A method to approximate DoR was proposed in [17]. It showed better performance, e.g, fewer refinement iterations, than [48] for the tested designs. The idea of using DoR for STE refinement was further extended by Adams in [3]. Adams showed how to compute DoR for STE property with symbolic indexing, and for arbitrary gates. Adams also explored alternatives to refining inputs by assigning variables. For example, we can introduce guards so that we drive the input with a variable only when the spurious counterexample happens. The results of using alternatives are mixed and the authors stressed the importance to keep the guard logic simple. All existing methods refine the abstraction by adding new variables to the antecedent. Each time a variable is added, the size of the set of input sequences doubles and the STE run is made closer to a full-scale costly binary symbolic simulation. In summary, aforementioned refinement approaches all modify the ternary stimulus for some signals to replace some X assignments with Boolean variables assignments. On the other hand, our refinement methods never modify the ternary stimulus. They modify the (unrolled) Boolean transition function instead.

Two existing SAT-implementations of STE were proposed by Roorda et al. [43] and Bjesse [8]. [43] does not use symbolic simulation as in [8]. Instead, it translates each ternary gate to constraints of inputs and output of the ternary gate. The authors showed that [43] is faster than [8] because it generates fewer clauses. We showed that our SAT based implementation of STE outperforms the implementation in [43] significantly.

CAMs have been extensively used as an example in related work about STE ([42, 43, 44, 48]). However, the CAMs verified are substantially different from ours, for example, tag and data pair is updated differently, and our CAM can delete a tag while theirs can’t. The property we verify is intuitive to write but challenging to verify because it covers both of the logic involved in writing/updating/deleting tag data and reading the data associated with the read tag. Their properties are small and inductive, and can be thought as a decomposition of our property. However, decomposition is well-known to be difficult, and making them inductive is often challenging. [42] used BDD to check satisfiability while we use SAT solvers. Moreover, the designs we use for evaluating our techniques go way beyond the conventional data-path designs used to study STE. The designs range from

a buffer design that is needed for processing traffic from real-world bus protocols to a synthetic design based on a superscalar microprocessor’s bypass path. Such designs have considerable complexity and we observe significant and sometimes orders of magnitudes improvement with our abstraction techniques.

The use of ternary domain and ternary abstraction of circuits can be viewed as a special case of multi-value model checking. Refer to Chechik and Gurfinkel et al. [16, 30] for theories and implementations of general multi-value model checking.

Chapter 3

Guard-value Encoding

In this chapter, we present our first contribution of the thesis, which is the guard-value encoding. In Section 3.1, we define the guard-value encoding and describe how it implements a ternary transition function with two Boolean transition functions called guard transition function and value transition. More importantly, in this section, we also present two properties of the guard-value encoding, which will be leveraged to make the guard-value encoding based ternary verification much faster than Boolean verification and ternary verification based on the dual-rail encoding. The key to achieve it is shown in Section 3.3 and 3.4, where we present optimizations. When it comes to symbolically representing a set of ternary states with a set of pairs of Boolean states, our optimizations help reduce the cardinality of the set of pairs of Boolean states as well as the AIG-size of the Boolean states' symbolic representation (Section 3.3). For a ternary transition function, our optimizations can reduce the cardinality of the set encoding the ternary states reachable in the next cycle and they can also reduce the AIG-size of the value transition function. We present its first application, which is memory abstraction. Lastly, this chapter is concluded with a summary.

3.1 Guard-value Encoding

With the guard-value encoding, a value from the ternary domain is represented as a pair of Boolean values (Definition 12). We will refer to the left of the pair as the “guard”, and the right of the pair as the “value”. The ternary value is $\mathbf{X}$ as long as its guard is 0: both $\langle 0, 0 \rangle$ and $\langle 0, 1 \rangle$ represent $\mathbf{X}$.

Definition 12 (Guard-value encoding). *In guard-value encoding, $\langle 1, 1 \rangle$ represents 1. $\langle 1, 0 \rangle$ represents 0. $\langle 0, 0 \rangle$ and $\langle 0, 1 \rangle$ both represent $\mathbf{X}$.*

As shown in Definition 12, both $\langle 0, 0 \rangle$ and $\langle 0, 1 \rangle$ encodes $\mathbf{X}$, which illustrates the flexibility in representing $\mathbf{X}$.

A ternary transition function $\hat{\text{Tr}}$ can be implemented with a pair of Boolean transition functions using guard-value encoding. The Boolean transition function computing the next state for signals' guard field is called the guard transition function (denoted by Tr_g) while the value transition function (denoted by Tr_v) computes the next state for signals' value field. Because we assume a hardware design to verify is represented as an AIG, ternary transition functions only involve ternary AND and NOT and it suffices to show how ternary AND and NOT translate to Boolean gates.

Table 3.1 and Table 3.2 show the guard and value of the output of a ternary AND and a ternary NOT respectively, where ‘-’ could be either 0 or 1.

Table 3.1: The guard and value of the output of a ternary AND ($\hat{\wedge}$)

$\hat{\wedge}$	$\langle 1, 0 \rangle$	$\langle 1, 1 \rangle$	$\langle 0, - \rangle$
$\langle 1, 0 \rangle$	$\langle 1, 0 \rangle$	$\langle 1, 0 \rangle$	$\langle 1, 0 \rangle$
$\langle 1, 1 \rangle$	$\langle 1, 0 \rangle$	$\langle 1, 1 \rangle$	$\langle 0, - \rangle$
$\langle 0, - \rangle$	$\langle 1, 0 \rangle$	$\langle 0, - \rangle$	$\langle 0, - \rangle$

Table 3.2: The guard and value of the output of a ternary NOT ($\hat{\neg}$)

$\hat{\neg}$	$\langle 1, 0 \rangle$	$\langle 1, 1 \rangle$	$\langle 0, - \rangle$
	$\langle 1, 1 \rangle$	$\langle 1, 0 \rangle$	$\langle 0, - \rangle$

Let the inputs to a ternary AND have values $\langle a_g, a_v \rangle$ and $\langle b_g, b_v \rangle$. Then based on Table 3.1, the output's value, denoted as $\langle \mathbf{o}_g, \mathbf{o}_v \rangle$ are:

$$\begin{aligned}\mathbf{o}_g &= (a_g \wedge b_g) \vee (a_g \wedge \neg a_v) \vee (b_g \wedge \neg b_v) \\ \mathbf{o}_v &= a_v \wedge b_v\end{aligned}\tag{3.1}$$

Similarly, the output's value of a ternary NOT gate can be computed in terms of its input's value, denoted as $\langle a_g, a_v \rangle$:

$$\begin{aligned}\mathbf{o}_g &= a_g \\ \mathbf{o}_v &= \neg a_v\end{aligned}\tag{3.2}$$

Just like dual-rail encoding, with Equation 3.1 and Equation 3.2, we also reduce symbolically simulating a ternary AND/NOT gate to Boolean symbolic simulations.

By using what we have so far, we can construct $\mathcal{M}_{gv}$ that encodes $\hat{\mathcal{M}}$ using guard-value encoding. This way, verifying $\hat{\mathcal{M}}$ can be reduced to verifying the Boolean transition function $\mathcal{M}_{gv}$. The number of signals in $\mathcal{M}_{gv}$ is twice as many as the number of signals in $\mathcal{M}$ because for any signal n from $\mathcal{M}$, its guard signal n_g and value signal n_v are in $\mathcal{M}_{gv}$: $S_{gv} = \{n_g, n_v : n \in S\}$.

$\mathcal{M}_{gv} = (\hat{Q}_{R_{gv}}, \text{Tr}_{gv}, \Sigma_{I_{gv}})$, where

- The set of initial states $\mathring{Q}_{R_{gv}}$ encodes $\hat{Q}$ using guard-value encoding: for any ternary initial state $\hat{q}$ from $\hat{Q}$, there is a corresponding Boolean initial state q from $\mathring{Q}_{R_{gv}}$ that encodes $\hat{q}$ and for any Boolean state in $\mathring{Q}_{R_{gv}}$ there is a ternary state in $\hat{Q}$ that is encoded by it.

We decompose $\mathring{Q}_{R_{gv}}$ with regard to its domain into $\mathring{Q}_{R_g}$ and $\mathring{Q}_{R_v}$, which are the initial states for only the guard signals and value signals respectively. That is, for any $q \in \mathring{Q}_{R_{gv}}$, there is $q_g \in \mathring{Q}_{R_g}$ and $q_v \in \mathring{Q}_{R_v}$ such that $\forall n \in S_R . q(\mathbf{n}_g) = q_g(\mathbf{n}_g)$ and $q(\mathbf{n}_v) = q_v(\mathbf{n}_v)$. We call q_g guard state and q_v value state.

- Transition function Tr_{gv} is constructed by composing the Boolean implementations of ternary AND and NOT found in $\hat{\text{Tr}}$. We discuss it more in the next section.
- Stimulus $\Sigma_{I_{gv}}$ encodes $\hat{\Sigma}_I$: for any ternary stimulus, there is a Boolean stimulus from $\Sigma_{I_{gv}}$ encoding it and for any Boolean stimulus, there is a ternary stimulus from $\hat{\Sigma}_I$ encoded by it. A Boolean stimulus $\pi_I \in \Sigma_{I_{gv}}$ encodes a ternary stimulus $\hat{\pi}_I$ iff $\forall t . \forall \mathbf{n} \in S_I . \langle \pi_I^t(\mathbf{n}_g), \pi_I^t(\mathbf{n}_v) \rangle = \hat{\pi}_I^t(\mathbf{n})$.
 Σ_{I_g} and Σ_{I_v} to denote the stimulus for only the guard signals and value signals respectively.

Lemma 4. *Every trace of $\mathcal{M}_{gv}$ encodes a ternary trace from $\hat{\mathcal{M}}$. As a result, $\mathcal{M}_{gv} \models (\mathbf{o}_g \wedge \mathbf{o}_v) \implies \hat{\mathcal{M}} \models (\mathbf{o} = 1)$. Recall that it is assumed that the property to verify is represented as a primary output $\mathbf{o}$ of the design.*

In the next section, we study the guard-value encoding of a ternary transition function in details and compare it with the dual-rail encoding of a ternary transition function comprehensively.

3.2 Comparing the Guard-value and Dual-rail Implementations of Ternary Transition Functions

The dual-rail transition function Tr_{dr} and Tr_{gv} are compared in terms of their AIG-size, the number of inputs and some other characteristics.

Both Tr_{dr} and Tr_{gv} are constructed by replacing the ternary AND and NOT with their corresponding ternary encodings in $\hat{\text{Tr}}$. The AIG-size of a transition function is the number of AND gates. Based on 2.4 and Formula 2.5 on page 24, using structural induction on the AIG, we can conclude the following facts about Tr_{dr} :

- For each signal, the transition functions for its high rail and low rail have the same size and it is equal to the original transition function: $\forall \mathbf{n} \in S_R . |\text{Tr}_{dr}(\mathbf{n}_h)| = |\text{Tr}_{dr}(\mathbf{n}_l)| = |\text{Tr}(\mathbf{n})|$.

- Let signal m be a signal among the inputs of $\text{Tr}(n)$. Then, m 's high rail (low rail) is among the inputs of $\text{Tr}_{dr}(n_h)$ if there is a path between m and n and there are an even (odd) number of NOT gates on that path. Similarly, Signal m 's high rail (low rail) is among the inputs of $\text{Tr}_{dr}(n_l)$ if there is a path between m and n and there are odd (even) number of NOT gates on that path. Therefore, the number of inputs for $\text{Tr}_{dr}(n_h)$ and $\text{Tr}_{dr}(n_l)$, denoted by $|\text{supp}(\text{Tr}_{dr}(n_h))|$ and $|\text{supp}(\text{Tr}_{dr}(n_l))|$ is between $|\text{supp}(\text{Tr}(n))|$ and $2 \times |\text{supp}(\text{Tr}(n))|$, where *supp* abbreviates "support".
- For any signal n , neither of $\text{Tr}_{dr}(n_h)$ and $\text{Tr}_{dr}(n_l)$ is likely to be equivalent to Tr , regardless of the mapping of the inputs. For example, Figure 2.2 from page 18 shows that neither of the high rail and low rail for a ternary multiplexer is a multiplexer.

Based on 3.1 and Formula 3.2, using structural induction on the AIG, we can conclude the following facts about Tr_{gv} :

- For each signal, the value transition function has the same size as the original transition function. Because $\vee$ translates to 1 AND gate and 2 NOT gates, the guard transition function has 4 times more AND gates than the original transition function: $\forall n \in S_R . |\text{Tr}_v(n)| = |\text{Tr}(n)|$ and $|\text{Tr}_g(n)| = 5 \times |\text{Tr}(n)|$.
- The transition function for a signal's value only depends on the value of its inputs, while the transition function for a signal's guard depends on both of the value and guard of its inputs. Therefore,
 - $|\text{supp}(\text{Tr}_g(n))| \geq |\text{supp}(\text{Tr}_{dr}(n_h))|$,
 - $|\text{supp}(\text{Tr}_g(n))| \geq |\text{supp}(\text{Tr}_{dr}(n_l))|$,
 - $|\text{supp}(\text{Tr}_v(n))| \leq |\text{supp}(\text{Tr}_{dr}(n_h))|$ and
 - $|\text{supp}(\text{Tr}_v(n))| \leq |\text{supp}(\text{Tr}_{dr}(n_l))|$.
- The transition function for a signal's value in the guard-value encoding is identical to the original transition function except the inputs. Mathematically, it is captured in Lemma 5. The guard transition function computes the care condition for that signal and it is unlikely to be the same as the original function.

Lemma 5. $\forall n \in S . \text{Tr}_v(n) = \text{Tr}(n)(m_v/m)$, where we use $\text{Tr}(n)(m_v/m)$ to denote replacing every input of $\text{Tr}(n)$, m , with its corresponding value signal m_v .

Corollary 1 is derived from Lemma 5. The corollary shows that when computing the value field of ternary traces, the guard field of the initial states and stimulus are not involved.

Corollary 1. Let $\mathcal{M}_v$ be $(\mathcal{Q}_{R_v}, \text{Tr}(m_v/m), \Sigma_{I_v})$. The set of signals in $\mathcal{M}_v$ is $\{n_v : n \in S\}$. Then, $\mathcal{L}(\mathcal{M}_v)$ includes exactly the value fields of the traces in $\mathcal{L}(\mathcal{M}_{gv})$. That is

$$1. \forall \sigma_v \in \mathcal{L}(\mathcal{M}_v) . \exists \sigma_{gv} \in \mathcal{L}(\mathcal{M}_{gv}) . \forall n \in S . \forall t . \sigma_v^t(n_v) = \sigma_{gv}^t(n_v) \text{ and}$$

$$2. \forall \sigma_{gv} \in \mathcal{L}(\mathcal{M}_{gv}) . \exists \sigma_v \in \mathcal{L}(\mathcal{M}_v) . \forall n \in S . \forall t . \sigma_v^t(n_v) = \sigma_{gv}^t(n_v)$$

The main disadvantage of guard value encoding is the size of guard transition functions. Fortunately, in practice, there often exists a great amount of logic redundancy, which presents opportunities for logic optimizations. We work on two toy examples (Example 3.2.1 and Example 3.2.2) to show the logic redundancy as well as how we can significantly simplify the guard transition functions. The logic redundancy comes from the way we construct guard transition functions and the fact that in practice, many signals cannot be X. Manual simplifications are powerful but not practical. Instead, we find that off-the-shelf logic optimizations [39, 41]. work very well at simplifying guard transition functions and signals' guard values. The empirical results can be found in Section 8.2.

Example 3.2.1. *In this example, we simplify the guard of a multiplexer with y and z as inputs and s as the select line. The step by step simplification is described below. First, we use Boolean AND and NOT to implement the ternary ones based on 3.1 and Formula 3.2. Next we do case-splitting on whether the select line has don't-care value and then simplify the if branch and else branch. In the 3rd step, we use Boolean gates to implement the ternary OR. In the end, we use if-then-else operator for brevity.*

$$\begin{aligned}
\text{guard of } (y \hat{\wedge} s) \hat{\vee} (z \hat{\wedge} \neg s) &\equiv \text{guard of } (\langle (y_g \wedge s_g) \vee (y_g \wedge \neg y_v) \vee (s_g \wedge \neg s_v), y_v \wedge s_v \rangle \hat{\vee} \\
&\quad \langle (z_g \wedge s_g) \vee (z_g \wedge \neg z_v) \vee (s_g \wedge s_v), z_v \wedge \neg s_v \rangle) \\
&\equiv s_g ? \text{guard of } (\langle y_g \vee \neg s_v, y_v \wedge s_v \rangle \hat{\vee} \langle z_g \vee s_v, z_v \wedge \neg s_v \rangle) : \\
&\quad \text{guard of } (\langle y_g \wedge \neg y_v, y_v \wedge s_v \rangle \hat{\vee} \langle z_g \wedge \neg z_v, z_v \wedge \neg s_v \rangle) \\
&\equiv s_g ? ((y_g \wedge s_v) \vee (z_g \wedge \neg s_v)) : \\
&\quad (\neg y_v \wedge y_g \wedge \neg z_v \wedge z_g) \\
&\equiv s_g ? (s_v ? y_g : z_g) : \\
&\quad (\neg y_v \wedge y_g \wedge \neg z_v \wedge z_g)
\end{aligned} \tag{3.3}$$

After simplification, there are 9 ANDs in the guard because besides 3 ANDs, there are two if-then-else and an if-then-else is implemented using 3 ANDs. Without any simplification, there would have been $3 \times 5 = 15$ ANDs.

Example 3.2.2. *Let $\mathbf{a} = (a[0], a[1], \dots)$ be a vector of signals, where the signal selected by a vector of k signals $\mathbf{d}$ is denoted by $\mathbf{a}[\mathbf{d}]$. The guard and value field of the vector $\mathbf{a}$ is denoted by $\mathbf{a}_g = (a_g[0], a_g[1], \dots)$ and $\mathbf{a}_v = (a_v[0], a_v[1], \dots)$ respectively. In this example, we show how the guard of $\mathbf{a}[\mathbf{d}]$, denoted by $\mathbf{a}_{d_g}$, can be significantly simplified when the signals in the vector $\mathbf{d}$ cannot have value X, i.e., their guards cannot be 0. We implement $\mathbf{a}[\mathbf{d}]$ with a chain of multiplexers:*

$$\mathbf{a}[\mathbf{d}] = d = 0 ? \mathbf{a}[0] : (d = 1 ? \mathbf{a}[1] : (d = 2 ? \mathbf{a}[2] : \dots))$$

As shown below, we iteratively use Formula 3.3 to compute the guard. Note that $\mathbf{d}_g$ is a vector of 1s, therefore we use the if-branch in Formula 3.3. After we complete computing

the guard for every branch, it becomes d_v selecting among a_g , i.e., $a_g[d_v]$.

$$\begin{aligned}
\text{guard of } a[d] &\equiv (d_v = 0) ? a_g[0] : \text{guard of } \{(d_v = 1) ? a[1] : [(d_v = 2) ? a[2] : \dots]\} \\
&\equiv (d_v = 0) ? a_g[0] : \{(d_v = 1) ? a_g[1] : \text{guard of } ((d_v = 2) ? a[2] : \dots)\} \\
&\equiv \dots \\
&\equiv a_g[d_v]
\end{aligned} \tag{3.4}$$

The guard-value encoding's flexibility in encoding X and its property described in Lemma 5 are the most important advantages over the dual-rail encoding. In the next two sections, we develop techniques to simplify symbolic states and ternary transition functions based on these two properties.

3.3 Value States Simplification using the Care Set

A pair of Boolean expressions $\langle e_g, e_v \rangle$ in the guard-value encoding represents the symbolic ternary value $e_g ? e_v : X$, which is X when e_g is 0. Therefore, e_v 's value only matters if $e_g = 1$. In other words, e_g characterizes the set of input assignments where we care about e_v 's value. Such a set is often referred to as the care set for a Boolean function, which has been extensively studied and used in logic optimization (e.g., [45, 38, 51]). It can be extended to simplifying a symbolic ternary state by simplifying the values that the state maps the signals to. Example 3.3.1 illustrates how we can use the care-set to either optimize the number of logic operators or the size of the state space.

Example 3.3.1. *Let us again consider the set of ternary states from Example 2.2.1: $\{(1, X, X), (X, 1, X), (X, X, 1), (0, 0, 0)\}$, which is symbolically represented by the ternary function vector:*

$$\begin{aligned}
&(\neg i_1 \wedge \neg i_0 ? 1 : (i_1 \wedge i_0 ? 0 : X) , \\
&\neg i_1 \wedge i_0 ? 1 : (i_1 \wedge i_0 ? 0 : X) , \\
&i_1 \wedge \neg i_0 ? 1 : (i_1 \wedge i_0 ? 0 : X))
\end{aligned} \tag{3.5}$$

In Table 3.3, we use different encodings to represent the set of 4 ternary states and to represent the symbolic ternary state. For each encoding, the first row lists the high rails (or guards) while the second row lists the low rails (or values). As already shown in Section 2.6, using dual-rail encoding, a symbolic ternary state is represented by two symbolic Boolean states, which are the high rail state and the low rail state, as shown in the 3rd column. Similarly, using guard-value encoding, a symbolic ternary state can be represented by two symbolic Boolean states, which will be called the guard state and the value state. They generate the set of guards and values respectively.

The symbolic high rail state (guard state) and symbolic low rail state (value state) are constructed from the concrete set of states in the same way as we construct the Formula 3.5

in Example 2.2.1. Let us construct the symbolic high rail state as an example. (i_1, i_0) is the indexing vectors we use to select among the array of 4 concrete high rail states. Signal a , b and c 's high rails are 1 for all of the 4 states' high rails except the last state. The last state's index is $i_1 \wedge i_0$. Therefore, the symbolic high rail state is $(\neg(i_1 \wedge i_0), \neg(i_1 \wedge i_0), \neg(i_1 \wedge i_0))$

For the guard-value encoding, by choosing between $\langle 0, 0 \rangle$ and $\langle 0, 1 \rangle$ for the encoding of $\mathbf{X}$, there could be numerous different sets of value states. We demonstrate it in Table 3.3, where we show 3 different sets of value states and label them with v , v^* , v^{**} respectively.

1. v is the baseline that is compared to v^* and v^{**} . We choose to use $\langle 0, 0 \rangle$ uniformly for encoding $\mathbf{X}$.
2. we use v^* to demonstrate that we can carefully choose the encoding for $\mathbf{X}$ to optimize the state space for the value state. In this example, it happens to be uniformly using $\langle 0, 1 \rangle$ to represent $\mathbf{X}$. The value state space is $\{(1, 1, 1), (0, 0, 0)\}$, which is the smallest among all possibilities.
3. we use v^{**} to demonstrate that, alternatively, we can minimize the number of logic operators in the symbolic value state. To minimize the number of and for this example, we need to use $\langle 0, 1 \rangle$ and $\langle 0, 0 \rangle$ alternatively to encode $\mathbf{X}$ appearing within the set of value states. For example, to minimize the size of the symbolic value for signal a , we use $\langle 0, 1 \rangle$ to represent $\mathbf{X}$ in the 2nd state for a and $\langle 0, 0 \rangle$ to represent $\mathbf{X}$ in the 3rd state for a . As a result, the symbolic value uses 0 AND which is not only the smallest among all possibilities for guard-value encoding, but also smaller than the high-rail and low-rail state in dual-rail encoding.

As demonstrated in the above example, the simplification can achieve value state space reduction or reduction in terms of the AIG-size of value states' symbolic representation. Even though it is often the case that smaller state space yields simpler symbolic representation, these two goals are orthogonal. It is important to consider both simplifications goals for the simplification to be effective for SAT-based formal verification.

Given a set of ternary states, we can compute the theoretical maximum value state space reduction as follows. The procedure shown in Listing 3.1 returns a set of ternary states, where for any pair of ternary states, there exists at least one signal that's assigned conflicting values (i.e., assigned 0 by one state and 1 by the other). Let the size of the returned set of ternary states be m . Then m is the bound of the max value state space reduction that the guard-value encoding can achieve (Lemma 6): no matter how use encode $\mathbf{X}$, the number of value states is always m . The pseudo-code has a complexity of $\mathcal{O}(n^2)$, where n is the number of ternary states, which makes it usually impractical in practice.

Listing 3.1: Maximum value state space reduction

```
valueStateSpaceRed ( $\hat{q}_0, \hat{q}_1, \dots, \hat{q}_{n-1}$ )
  processed = {}
```

Table 3.3: Comparison of encodings
Concrete states

Index		$\neg i_1 \wedge \neg i_0$	$\neg i_1 \wedge i_0$	$i_1 \wedge \neg i_0$	$i_1 \wedge i_0$
Variables		(a, b, c)	(a, b, c)	(a, b, c)	(a, b, c)
State		$\{(1, X, X),$	$(X, 1, X),$	$(X, X, 1),$	$(0, 0, 0)\}$
DR	h	$\{(1, 1, 1),$	$(1, 1, 1),$	$(1, 1, 1)$	$(0, 0, 0)\}$
	l	$\{(0, 1, 1),$	$(1, 0, 1),$	$(1, 1, 0)$	$(1, 1, 1)\}$
GV	g	$\{(1, 0, 0),$	$(0, 1, 0),$	$(0, 0, 1),$	$(1, 1, 1)\}$
	v	$\{(1, \emptyset, \emptyset),$	$(\emptyset, 1, \emptyset),$	$(\emptyset, \emptyset, 1),$	$(0, 0, 0)\}$
	v*	$\{(1, \textcolor{red}{1}, \textcolor{red}{1}),$	$(\textcolor{red}{1}, 1, \textcolor{red}{1}),$	$(\textcolor{red}{1}, \textcolor{red}{1}, 1),$	$(0, 0, 0)\}$
	v**	$\{(1, \textcolor{red}{1}, \textcolor{red}{1}),$	$(\textcolor{red}{1}, 1, \emptyset),$	$(\emptyset, \emptyset, 1),$	$(0, 0, 0)\}$

		Symbolic state		
		a	b	c
DR	h	$\neg(i_1 \wedge i_0)$	$\neg(i_1 \wedge i_0)$	$\neg(i_1 \wedge i_0)$
	l	$\neg(\neg i_1 \wedge \neg i_0)$	$\neg(\neg i_1 \wedge i_0)$	$\neg(i_1 \wedge \neg i_0)$
	g	$(\neg i_1 \wedge \neg i_0) \vee (i_1 \wedge i_0)$	i_0	i_1
GV	v	$\neg i_1 \wedge \neg i_0$	$\neg i_1 \wedge i_0$	$i_1 \wedge \neg i_0$
	v*	$\neg(i_1 \wedge i_0)$	$\neg(i_1 \wedge i_0)$	$\neg(i_1 \wedge i_0)$
	v**	$\neg i_1$	$\neg i_1$	$\neg i_0$

```

unprocessed = { $\hat{q}_0, \hat{q}_1, \dots, \hat{q}_{n-1}$ }
reducedStates = {}
for  $\hat{q}_i$  in unprocessed:
    move  $\hat{q}_i$  from unprocessed to processed
    for  $\hat{q}_j$  in unprocessed:
        if  $\hat{q}_i \sqcap \hat{q}_j \neq \perp$ :
             $\hat{q}_i = \hat{q}_i \sqcap \hat{q}_j$ 
            move  $\hat{q}_j$  from unprocessed to processed
    add  $\hat{q}_i$  to reducedStates
return reducedStates

```

Lemma 6. *For the set of ternary states $\{\hat{q}_0, \hat{q}_1, \dots, \hat{q}_{n-1}\}$, the minimum number of value states is the number of ternary states in $\text{valueStateSpaceRed}(\hat{q}_0, \hat{q}_1, \dots, \hat{q}_{n-1})$.*

Proof. Two ternary states, $\hat{q}_0$ and $\hat{q}_1$, can share the same value state iff they don't have conflicting assignment, i.e., $\hat{q}_0 \sqcap \hat{q}_1 \neq \perp$. The function $\text{valueStateSpaceRed}$ combines all ternary states that are not conflicted with q_i into 1 ternary state and adds it to the set of ternary states to return. Let us assume that there exists smaller set of value states. Then there must be two ternary states that are not conflicted with each other but not combined in $\text{valueStateSpaceRed}$, which is not possible. $\square$

The value state simplification using the care-set will be further developed in Section 5, where it is used to reduce formal verification complexity: provided human's knowledge of when an input or register signal's value becomes irrelevant to the verification problem, this technique is used to simplify the stimuli and initial state.

3.4 Value Transition Function Simplification using the Care Set

When a signal's relevancy to the verification result is dependent on other signals' values and/or Boolean variables, we can leverage it to simplify the signal's value transition function. We use $\hat{g}$ to denote a ternary expression that takes signals and Boolean variables as inputs and represents the condition under which signal n 's value is relevant to the verification. We can construct a new ternary transition function for n (denoted by $\hat{\hat{\text{Tr}}}(n)$):

$$\hat{\hat{\text{Tr}}}(n) = \hat{g} ? \hat{\text{Tr}}(n) : \mathbf{X} \quad (3.6)$$

It is easy to see that $\hat{\hat{\text{Tr}}}(n)$ is an abstraction of n 's original ternary transition function $\hat{\text{Tr}}(n)$ because given the same current state, $\hat{\hat{\text{Tr}}}(n)$ maps n to the same value as $\hat{\text{Tr}}(n)$ when $\hat{g} = 1$ and otherwise, n is mapped to $\mathbf{X}$ in the next state.

Using the guard-value encoding, $\hat{\text{Tr}}(\mathbf{n})$'s corresponding value and guard transition function are:

$$\tilde{\text{Tr}}_v(\mathbf{n}_v) \equiv g_v ? \text{Tr}_v(\mathbf{n}_v) : \text{Tr}', \text{ where } \text{Tr}' \text{ can be any Boolean function.} \quad (3.7)$$

$$\begin{aligned} \tilde{\text{Tr}}_g(\mathbf{n}_g) &\equiv \neg g_g ? 0 : (g_v \wedge g_g ? \text{Tr}_g(\mathbf{n}_g) : 0) \\ &\equiv g_g \wedge g_v \wedge \text{Tr}_g(\mathbf{n}_g) \end{aligned} \quad (3.8)$$

(3.7) is the result of applying Lemma 5 and encoding $\mathbf{X}$ with $\langle 0, \text{Tr}' \rangle$. (3.8) is the result of applying Formual (3.3). The goal of this simplification is to make $\tilde{\text{Tr}}_v(\mathbf{n}_v)$ simpler in terms of the number of reachable states in the next cycle or/and $\tilde{\text{Tr}}_v(\mathbf{n}_v)$'s AIG-size. To achieve these goals,

1. we need to construct Tr' carefully and
2. we also need to come up with $\hat{g}$ such that $\text{Tr}_v(n)$ can be greatly simplified by restricting its domain based on $\hat{g} = 1$

In practice, we usually construct $\hat{g}$ in such a way such that it can never be $\mathbf{X}$. In such scenario, we may also denote it as g instead $\hat{g}$. One application of the value transition function simplification is memory abstraction. We will show a basic memory abstraction in the next section and a more advanced one in Section 6.

3.4.1 Memory Abstraction by Value Transition Function Simplification

In memory intensive hardware designs, the sheer number of memory slots could be overwhelming for formal verification. Fortunately, usually only a small number of memory slots affect the truth of the property to verify. For example, when we verify that a read operation returns the data in the memory at the read address, we only care about that one slot whose address matches the read address. Let the cared slot have an arbitrary address that is equal to $\vec{d}$, which is a vector of Boolean variables. Then, a better way to state the same property is that reading from the address $\vec{d}$ returns the data in the memory at address $\vec{d}$. This way of rewriting a property has been discussed in details as a part of the memory abstraction in [6]. In this thesis, we assume that the property is already written in a way that facilitates our memory abstraction and we focus on how to modify the value transition functions for reading and writing to the memory to reduce the state space of the memory. Our technique utilizes ternary domain, the guard-value encoding and the value transition function simplification introduced before. In this section, we deal with the memories that have a small number of memory slots that matter to the verification and these memory slots can be identified by addresses. These memories are exactly the ones that can be abstracted by the technique from [6]. Besides, our technique works in ternary domain while theirs works in the Boolean domain. Another interesting distinction is that

we achieve the state space reduction without actually reducing the number of memory slots as in [6]. In Chapter 6, we will generalize our technique to make it applicable to virtually any array of elements, which is a significant improvement over the existing work.

For simplicity of introducing our technique, let us consider a memory that can be reduced to just 1 slot and its address be $\vec{d}$.

Memory write

We use $\mathbf{wr}$ to denote write operation and its value field is $\mathbf{wr}_v$. The write address signal is $\mathbf{waddr}$ and the write data is $\mathbf{wdata}$. The signal $\mathbf{mem}[i]$ is the data in the i -th slot. Then, the transition function of the write operation, i.e., $\text{Tr}(\mathbf{mem}[i])$, is:

$$(\mathbf{wr} \wedge \mathbf{waddr} = i) ? \mathbf{wdata} : \mathbf{mem}[i] .$$

Based on Lemma 5, $\mathbf{mem}[i]$'s value transition function is simply:

$$(\mathbf{wr}_v \wedge \mathbf{waddr}_v = i) ? \mathbf{wdata}_v : \mathbf{mem}_v[i] .$$

Recall that to use value transition function simplification, we need to construct $\hat{g}$ and Tr' . For signal $\mathbf{mem}_v[i]$, $\hat{g}$ is $i = \vec{d}$ because we care about the value of the slot iff its address matches $\vec{d}$. We abstract the original transition function for $\mathbf{mem}[i]$ (i.e., $\text{Tr}(\mathbf{mem}[i])$) with the following ternary transition function

$$i = \vec{d} ? \hat{\text{Tr}}(\mathbf{mem}[i]) : \mathbf{X}$$

We will construct Tr' as we simplify the value transition function with $\hat{g}$ in steps from (3.9) to (3.12). In the first step, we are able to replace i with $\vec{d}$ in the if-branch because $i = \vec{d}$. In the second step, we use the if-branch operand to instantiate Tr' . In the end, the new value transition function for the i -th slot becomes $(\mathbf{wr}_v \wedge \mathbf{waddr}_v = \vec{d}) ? \mathbf{wdata}_v : \mathbf{mem}_v[\vec{d}]$, which is independent of i . In conclusion, every memory slot shares the same transition function for its value field. As a result, effectively, the memory is reduced to just 1 slot in $\mathcal{M}_v$.

$$\tilde{\text{Tr}}_v(\mathbf{mem}[i]) = (i = \vec{d}) ? ((\mathbf{wr}_v \wedge \mathbf{waddr}_v = i) ? \mathbf{wdata}_v : \mathbf{mem}_v[i]) : \text{Tr}' \quad (3.9)$$

$$\equiv (i = \vec{d}) ? ((\mathbf{wr}_v \wedge \mathbf{waddr}_v = \vec{d}) ? \mathbf{wdata}_v : \mathbf{mem}_v[\vec{d}]) : \text{Tr}' \quad (3.10)$$

$$\begin{aligned} &\equiv (i = \vec{d}) ? ((\mathbf{wr}_v \wedge \mathbf{waddr}_v = \vec{d}) ? \mathbf{wdata}_v : \mathbf{mem}_v[\vec{d}]) : \\ &\quad ((\mathbf{wr}_v \wedge \mathbf{waddr}_v = \vec{d}) ? \mathbf{wdata}_v : \mathbf{mem}_v[\vec{d}]) \end{aligned} \quad (3.11)$$

$$\equiv (\mathbf{wr}_v \wedge \mathbf{waddr}_v = \vec{d}) ? \mathbf{wdata}_v : \mathbf{mem}_v[\vec{d}] \quad (3.12)$$

Based on (3.8), the new guard transition function for $\mathbf{mem}[i]$ becomes (3.13) and it can be simplified to (3.14).

$$\tilde{\text{Tr}}_g(\mathbf{mem}[i]) \equiv i = \vec{d} \wedge \text{Tr}_g(\mathbf{mem}_g[i]) \quad (3.13)$$

$$\equiv i = \vec{d} \wedge \text{Tr}_g(\mathbf{mem}_g[\vec{d}]) \quad (3.14)$$

Let c_g and c_v denote the guard and value of $\mathbf{wr} \wedge (\mathbf{waddr} \hat{=} \vec{d})$ respectively. Usually, the guards of the input signals $\mathbf{wr}$, $\mathbf{waddr}$ and $\mathbf{wdata}$ are 1s, therefore $c_g = 1$. Then, based on our previous result on the guard of a mux (3.3), we can expand $\text{Tr}_g(\text{mem}_g[\vec{d}])$ as

$$\begin{aligned} \text{Tr}_g(\text{mem}_g[\vec{d}])) &= c_v ? \mathbf{wdata}_g : \text{mem}_g[i] \\ &= c_v ? 1 : \text{mem}_g[i] \end{aligned}$$

If initially $\text{mem}[\vec{d}]$ has non-X values, then based on the above formula, it's easy to see that it will have Boolean value in every cycle, i.e., $\text{Tr}_g(\text{mem}_g[\vec{d}]) = 1$. This way, (3.14) is further simplified to just $i = \vec{d}$.

Memory read

Let $\mathbf{raddr}$ be the read address. The original transition function of $\mathbf{rdata}$ (read data) is $\text{mem}[\mathbf{raddr}]$ (i.e., $\text{Tr}(\mathbf{rdata}) = \text{mem}[\mathbf{raddr}]$). Because we are only interested in the read data if the read address matches $\vec{d}$, we can construct $\hat{g}$ as $\mathbf{raddr} \hat{=} \vec{d}$ to use in the new ternary transition function of $\mathbf{rdata}$:

$$(\mathbf{raddr} \hat{=} \vec{d}) ? \hat{\text{Tr}}(\mathbf{rdata}) : \mathbf{X}$$

The corresponding new value transition function is:

$$(\mathbf{raddr}_v = \vec{d}) ? \text{mem}_v[\mathbf{raddr}_v] : \text{Tr}'$$

We simplify it in a similar fashion as the write operation and use $\text{mem}_v[\vec{d}]$ as Tr'

$$\begin{aligned} \tilde{\text{Tr}}_v(\mathbf{rdata}_v) &\equiv (\mathbf{raddr}_v = \vec{d}) ? \text{mem}[\mathbf{raddr}] : \text{Tr}' \\ &\equiv (\mathbf{raddr}_v = \vec{d}) ? \text{mem}_v[\vec{d}] : \text{mem}_v[\vec{d}] \\ &\equiv \text{mem}_v[\vec{d}] \end{aligned}$$

The simplified value transition function for $\mathbf{rdata}$ becomes independent of $\mathbf{raddr}$.

Same as $\mathbf{waddr}$, the input signal $\mathbf{raddr}$ is not assigned X in practice. This way, based on (3.8), the new guard transition function for $\mathbf{rdata}$ becomes

$$(\mathbf{raddr}_v = \vec{d}) \wedge \text{Tr}_g(\mathbf{rdata}), \text{ where } \text{Tr}_g(\mathbf{rdata}) \text{ is the old guard transition function.}$$

Also because $\mathbf{raddr}$ cannot be X, $\text{Tr}_g(\mathbf{rdata})$ is just $\text{mem}_g[\mathbf{raddr}_v]$ (recall (3.4)), which simplifies $\mathbf{rdata}$'s new guard transition function to:

$$(\mathbf{raddr}_v = \vec{d}) \wedge \text{mem}_g[\vec{d}]$$

Furthermore, recall that $\text{Tr}_g(\text{mem}_g) = 1$ when the guards of the input signals $\mathbf{wr}$, $\mathbf{waddr}$ and $\mathbf{wdata}$ are 1s and initially $\text{mem}[\vec{d}]$ have Boolean values. This way, the new guard

transition function for `rdata` is simply $\text{raddr}_v = \vec{d}$, which can be interpreted as when the read address is not equal to $\vec{d}$, the read data is `X` and otherwise, it is the data that is written to the memory at location $\vec{d}$.

We have shown that the value transition functions of both memory read and write are identical among the memory slots, which effectively reduces the number of memory slots to 1. Our memory abstraction makes the new guard transition functions more complicated because of the inclusion of $\hat{g}$. But we have shown that the new guard transition functions can be greatly simplified in practice. This technique can be easily extended to cases where the memory slots of more than 1 address matter to the verification result. Beyond that, there are more advanced application of value transition function simplification in memory abstraction, which we will show in Section 6.

3.5 Summary

In this chapter, we introduce the guard-value encoding and present how it can encode ternary states, ternary transition function and eventually ternary transition systems. We study and compare the AIG-size, the number of supports and other characteristics of ternary transition functions implemented with the guard-value encoding and the dual-rail encoding. More importantly, we show that the guard-value encoding has two desirable properties that include the flexibility in encoding `X` and that its value transition function is identical to the original Boolean transition function. We leverage these two properties to develop the techniques that can simplify value states and value transition functions. As an application of the value transition function simplification, we show how to use it to achieve memory abstraction. These two techniques will be further developed in Chapter 5 and Chapter 6. But first, let us analyze and present the advantages of the guard-value encoding over the dual-rail encoding in the context of formal verification of a transition system in the next chapter.

Chapter 4

Effective Verification with Ternary Domain

In this chapter, we explain why verifying a ternary abstraction of a Boolean transition system with the guard-value encoding is faster than using the dual-rail encoding or verifying the Boolean transition system directly in the Boolean domain. This chapter will be divided into 4 sections. We first introduce how we compare them and the measures we consider. We assume SAT solvers are used. Then, we show that intuitively these three approaches are solving the same problem differently. Next, we explain why the ternary verification with the guard-value encoding is faster than using the dual-rail encoding and the Boolean verification respectively. Lastly, we present other useful advantages of ternary verification with the guard-value encoding.

4.1 Introduction

For the ease of discussion, we assume that the design and the property to verify are combined into an AIG with a single primary output. This way, the design satisfies the property iff the output is 1 for any clock cycle (i.e., constant 1). For anything that needs to check the satisfiability during verification, we will loosely refer to it as a verification problem. For example, BMC generates a verification problem for each cycle and induction generates two verification problems, which are for base step and inductive step respectively. One benefit of using the AIG as data structure is that there is a wide range existing AIG-based logic optimization techniques available. AIGs are translated to Conjunctive Normal Form (CNF) using efficient methods such as [23], which will then be solved by SAT solvers. Because induction can also be reduced to BMC as we have shown in Section 2.4.2, we will only discuss ternary BMC and regular BMC.

Firstly, ternary BMC based on dual-rail encoding (BMC-DR) and guard-value encoding (BMC-GV) as well as Boolean BMC can be regarded as 3 distinct encodings of the same

verification problem:

- BMC checks that the output is always 1 in the k -th cycle.
- Ternary simulation using dual-rail encoding computes the “may be 1” condition (high rail) and “may be 0” (low rail) condition for each signal. BMC-DR checks that the output may not be 0 (low rail is 0) in the k -th cycle. Unless there could be over-constrained, the high rail need not be checked (same as Symbolic Trajectory Evaluation [46]). When over-constraints happen, symbolic simulation is usually stopped.
- Ternary simulation using guard-value encoding complements Boolean simulation: it not only computes the value for each signal but also the care condition (i.e., guard), which indicates the condition when the change of value at this signal affects the primary output (i.e., the verification property). BMC-GV verifies the guard and value of the primary output: 1) guard verification: checking that X cannot propagate to the output, that is the primary output’s guard must be 1 and, 2) value verification: checking that BMC with modified stimulus and initial state (recall $\mathcal{M}_v$ from Corollary 1) passes for the cycle k .

In order for the comparison to be fair, we use BMC, BMC-DR and BMC-GV to verify the same property, and use the same transition function¹ except that the gates are given ternary semantics for BMC-DR and BMC-GV. In order for the comparison to be meaningful, we use different initial state and stimulus for ternary BMC and BMC because as stated in Lemma 1, the initial state and stimulus define the abstraction level for ternary BMC. For example, if we use the same initial state and stimulus as BMC, ternary BMC behaves exactly the same as BMC, i.e., it will generate the same verification problem to SAT solvers. In order to attain soundness, the initial state and stimulus of ternary BMC are required to be an abstraction of BMC’s, i.e.,

$$\hat{\mathcal{Q}}_R \leq \hat{\mathcal{Q}}_R, \text{ and } \Sigma_I \leq \hat{\Sigma}_I$$

In Section 8.2, experimental results will show that BMC-GV is better than BMC-DR and BMC. In this section, we will make effort to explain the concepts behind empirical results. Most of the presentations will be argumentative and about our intuitions. The quantitative measures we use to compare BMC, BMC-DR and BMC-GV are the size of AIGs and the state space. The size of AIG will affect the size of CNF given to SAT solvers. Note that by no means we imply that the size of AIGs and the state space can reliably predict the hardness of a SAT problem. But these are useful measures that have been used and discussed in literature. For example, Roorda [43] compares the proposed SAT based STE implementation with existing ones in terms of the size of CNF. Een, Mishchenko et al. [23, 40] show circuit-based approaches to reduce the size of CNF which

¹We may need to modify it slightly for refinement for ternary BMC.

leads to improving SAT solving performance. As for a smaller state space, it not only improves the worst-case performance, but also often significantly accelerates the verification in practice, as seen with many successful techniques that are able to reduce state space, such as symmetry reduction, partial order reduction and abstraction.

The comparisons among BMC, BMC-DR and BMC-GV are summarized in Table 4.1. They are compared in terms of:

- **size before logic opt:** it is the size of the verification problem before any logic optimization. Therefore, this measure only depends on the size of stimulus and the size of the transition functions.
- **state space:** input state space and reachable states
- **amenable to logic opt:** the more a verification problem is amenable to logic optimization, the less time logic optimization takes and the more logic optimization accelerates the verification.
- **comments:** it is where to write down any other important differences.

A set of Boolean states can be compactly abstracted by a set of ternary states. For example, a set of 4 Boolean states $\{(0, 0, 1), (0, 1, 1), (1, 0, 1), (1, 1, 1)\}$ can be abstracted by a set of just 1 ternary state $\{(X, X, 1)\}$. The lower bound of the number of variables needed to symbolically represent a set of k elements is $\lceil \log(k) \rceil$. Therefore, using ternary domain has the advantage of using fewer variables, which is a key driver for BDD-based STE's success.

4.2 BMC-DR and BMC-GV

Before we compare the size of verification problems for BMC-DR and BMC-GV, let us compare the sizes of the initial state and stimulus for DR and GV, since the gates used to represent the initial state and stimulus are most likely to be part of the verification problem. We only compare the size of the initial state here, since the comparison for stimulus can be done very similarly. Let us assume that in the initial state, the most compact representation for signal n 's value using dual-rail encoding is $\langle f_h, f_l \rangle$. Then using guard-value encoding and by leveraging the flexibility provided by the guard, n 's value can be encoded in GV with any pair of Boolean functions $\langle \neg(f_h \wedge f_l), f \rangle$ as long as $\neg(f_h \wedge f_l) \implies (f = f_h)$. f_h and $\neg f_l$ are the 2 straightforward candidates for f ². As a conclusion, the initial state in guard-value encoding is at least as compact as in the dual rail encoding and potentially can be significantly smaller through simplification with care-set.

²Here we assume that there is no over-constraints, i.e., f_h and f_l cannot both be 0

Besides the size of the initial state and stimulus, the size of verification problem is also determined by the size of the transition function as well as the inputs of the transition function. We will show that the differences in the inputs of the transition functions between two encodings make value verification problem much smaller than the verification problem of BMC-DR in practice.

For each non-input signal n , its value transition function $\text{Tr}_v(n)$ has the same size as its low rail transition function $\text{Tr}_{dr}(n_l)$ and its high rail transition function $\text{Tr}_{dr}(n_h)$. Let the inputs of the original transition for n be $\vec{in}$ and $\vec{r}$, i.e., $\text{Tr}(n)(\vec{in}, \vec{r})$, where $\vec{in} \subseteq S_I$ is a vector of some input signals and $\vec{r} \subseteq S_R$ is a vector of some non-input signals. Recall that in Section 3.2, we show that the inputs of $\text{Tr}_v(n)$ are the value fields of the original inputs, i.e., $\text{Tr}_v(n)(\vec{in}_v, \vec{r}_v)$, while the inputs of $\text{Tr}_{dr}(n_l)$ (or $\text{Tr}_{dr}(n_h)$) could include both rails. Specifically, both rails of signal m from $\vec{in}$ or $\vec{r}$ will appear in $\text{Tr}_{dr}(n_l)$ if there are at least two different paths between n and m in the AIG of $\text{Tr}(n)$ such that there are even number of NOT on one path and odd number on another.

Let n be the primary output. Then, the value verification problem at cycle t is

$$\text{Tr}_v(n) \left(\Sigma_{I_v}^{t-1}(\vec{in}), \Sigma_{R_v}^{t-1}(\vec{r}_v) \right) \quad (4.1)$$

$\Sigma_{I_v}^{t-1}(\vec{in})$ is the value field of the symbolic stimulus for the input vector in cycle $t - 1$ and $\Sigma_{R_v}^{t-1}(\vec{r})$ is the value field of the symbolic state of registers in cycle $t - 1$, which can be computed with symbolic simulation (2.1).

In order to describe the inputs to the BMC-DR verification problem, which is $\text{Tr}_{dr}(n_l)$, let us first partition $\vec{in}$ into $\vec{in}^{br}$ and $\vec{in}^{sr}$, where $\vec{in}^{br}$ includes the signals, whose both of high and low rails appear in the inputs and $\vec{in}^{sr}$ includes the signals, whose single rail appears in the inputs. Without loss of generality, we assume that it is the low rail. Similarly, $\vec{r}$ is partitioned into $\vec{r}^{br}$ and $\vec{r}^{sr}$. Then, The verification problem for BMC-DR at cycle t is

$$\begin{aligned} \text{Tr}_{dr}(n_l) \left(\Sigma_{I_{dr}}^{t-1}(\vec{in}_l^{sr}), \Sigma_{R_{dr}}^{t-1}(\vec{r}_l^{sr}), \right. \\ \Sigma_{I_{dr}}^{t-1}(\vec{in}_h^{br}), \Sigma_{I_{dr}}^{t-1}(\vec{in}_l^{br}), \\ \left. \Sigma_{R_{dr}}^{t-1}(\vec{r}_h^{br}), \Sigma_{R_{dr}}^{t-1}(\vec{r}_l^{br}) \right) \end{aligned} \quad (4.2)$$

Formula (4.2) can be simplified by replacing $\Sigma_{I_{dr}}^{t-1}(\vec{in}_l^{sr})$ and $\Sigma_{I_{dr}}^{t-1}(\vec{in}_l^{br})$ with $\Sigma_{I_{dr}}^{t-1}(\vec{in}_l)$ because $\vec{in}^{br} \cup \vec{in}^{sr} = \vec{in}$. Similarly, $\Sigma_{R_{dr}}^{t-1}(\vec{r}_l^{sr})$ and $\Sigma_{R_{dr}}^{t-1}(\vec{r}_l^{br})$ are replaced with $\Sigma_{R_{dr}}^{t-1}(\vec{r}_l)$. After simplification, the formula becomes:

$$\text{Tr}_{dr}(n_l) \left(\Sigma_{I_{dr}}^{t-1}(\vec{in}_l), \Sigma_{R_{dr}}^{t-1}(\vec{r}_l), \Sigma_{I_{dr}}^{t-1}(\vec{in}_h^{br}), \Sigma_{R_{dr}}^{t-1}(\vec{r}_h^{br}) \right) \quad (4.3)$$

Comparing Formula (4.1) for GV and Formula (4.3) for DR, though both encodings' transition functions have the same size, i.e., $|\text{Tr}_v(n)| = |\text{Tr}_{dr}(n_l)|$, it is clear that there are more gates in the expression for a node at time t in DR (Formula (4.3)) than that in GV because:

1. $\overrightarrow{\text{in}}^{br}$ or $\overrightarrow{\text{r}}^{br}$ is rarely empty, i.e., there exists at least one signal whose both rails are among the inputs of $\text{Tr}_{dr}(n_l)$. Or
2. there exists a signal in $\overrightarrow{\text{r}}$, let it be $\mathbf{m}$, such that *i*) $\mathbf{m}_h$ (or $\mathbf{m}_l$) is among the inputs of $\text{Tr}_{dr}(n_l)$, and *ii*) the inputs of $\text{Tr}_{dr}(\mathbf{m}_h)$ (or $\text{Tr}_{dr}(\mathbf{m}_l)$) include both rails of a signal. Therefore, the size of the symbolic value for $\mathbf{m}_l$ is larger than that of $\mathbf{m}_v$ for any cycle. When there are many signals that are like $\mathbf{m}$ among the inputs of $\text{Tr}_{dr}(n_l)$, it will result in the size of the state of the support signals in cycle $(t - 1)$ being much larger than those in GV, i.e., $(|\Sigma_{R_{dr}}^{t-1}(\overrightarrow{\text{r}})| + |\Sigma_{R_{dr}}^{t-1}(\overrightarrow{\text{r}}^{br})|) \gg (|\Sigma_{R_v}^{t-1}(\overrightarrow{\text{r}})| + |\Sigma_{R_v}^{t-1}(\overrightarrow{\text{r}}^{br})|)$.

In practice, 1) and 2) are extremely common, which makes the size of the value verification problem much smaller than BMC-DR. Furthermore, by using value simplification on the stimulus and initial state of BMC-DR, the value verification problem can be made even smaller. Some experimental results can be found in Section 8.2.

BMC-GV additionally has to check that $\mathbf{X}$ cannot propagate to the output (guard verification), which is a bigger verification problem than BMC-DR verifying that the output may not be 0, because $\text{Tr}_g(n)$ has more AND gates than $\text{Tr}_{dr}(n_l)$. Luckily, as discussed in Section 3.2, guards are amenable to logic optimizations. Besides, here we argue that intuitively, verifying that $\mathbf{X}$ cannot propagate to the output is an easier problem than verifying that the output cannot be 0, because in practice, most input signals' guards are 1s (i.e., they cannot be $\mathbf{X}$). Our experimental results support our intuition because most of the time, the guards are readily solved by logic optimizations.

In terms of state space, the value verification of BMC-GV could also be significantly smaller than verifying the low rail in BMC-DR. Example 3.3.1 demonstrates it well. According to the 3rd row in Table 3.3, the input space for the value component is $\{(1, 1, 1), (0, 0, 0)\}$ while the input space for the low rail verification is $\{(0, 1, 1), (1, 0, 1), (1, 1, 0), (1, 1, 1)\}$, if we assume the best case for BMC-DR where only the low rails of the inputs are used. In Section 5, we will further elaborate this idea. A welcoming side effect of reduction in the input state space is that it also makes value verification problem amenable to logic optimizations, that is, the value verification can be significantly accelerated by spending a relatively small amount of time for logic optimizations. Besides the state space reduction at the initial state and stimulus, we have also seen how BMC-GV can reduce the state space by memory abstraction in Section 3.4.1.

Lastly, for guard-value encoding, the value transition function is identical to the original transition function except the inputs, which is another advantage BMC-GV has over BMC-DR (Section 3.2).

4.3 BMC-GV and BMC

Ternary abstraction naturally enables us to compactly abstract a set of Boolean states with a set of ternary states regardless of the encoding used. For example, a set of 4 Boolean

Table 4.1: Comparing the value verification and guard verification of BMC-GV against BMC-DR and BMC

BMC-GV	criteria	BMC-DR	BMC
value verification	size before logic opt	++	$\approx$
	state space	++	++
	amenable to logic opt	++	++
	comments	(1)	
guard verification	size before logic opt	—	—
	state space	$\approx$	$\approx$
	logic opt	++	++
	comments	(2)	(3)

- ++: much better. —: worse. —: much worse. $\approx$: similar. For example, the first row reads “value verification is much better than BMC-DR in terms of the size before logic optimizations. Value verification has similar size as BMC before logic optimizations.”
- (1): value transition function is identical to the original transition function except the inputs.
- (2): guard verification verifies that X cannot propagation to the output and BMC-DR verifies that the primary output may not be 0. These are distinct verification problems. We argue that it is often easier to verify the X propagation problem in this paper.
- (3): BMC verifies that the primary output must be 1. It is a distinct verification problem from guard verification.

states $\{(0, 0, 1), (0, 1, 1), (1, 0, 1), (1, 1, 1)\}$ can be abstracted by a set of just 1 ternary state $\{(X, X, 1)\}$. The lower bound of the number of variables needed to symbolically represent a set of k elements is $\lceil \log(k) \rceil$. Therefore, using ternary domain has the advantage of using fewer variables, which is a key driver for BDD-based STE’s success. This section explains in details why BMC-GV can outperform BMC.

The value verification of BMC-GV is essentially BMC on the value transition system $\mathcal{M}_v = (\mathring{Q}_{R_v}, Tr(m_v/m), \Sigma_{I_v})$ which is defined in Corollary 1. $\mathcal{M}_v$ and $\mathcal{M}$ share the same transition function Tr . Therefore, before any logic optimization, the size of value verification (i.e., BMC on $\mathcal{M}_v$) and BMC are similar in sizes. In order to make BMC faster on $\mathcal{M}_v$ than $\mathcal{M}$, we use the flexibility provided by don’t-care to

1. make $\mathring{Q}_{R_v}$ and Σ_{I_v} have smaller state space than $\mathring{Q}_R$ and Σ_I and/or
2. make $\mathring{Q}_{R_v}$ and Σ_{I_v} have smaller size than $\mathring{Q}_R$ and Σ_I and/or

3. apply memory abstractions.

2 and 3 make $\mathcal{M}_v$ contains fewer traces than $\mathcal{M}$: $\mathcal{L}(\mathcal{M}_v) \subset \mathcal{M}$.

In Example 3.3.1, $\{(1, X, X), (X, 1, X), (X, X, 1), (0, 0, 0)\}$ abstracts the set of all Boolean states. If the Boolean domain is used, (v_2, v_1, v_0) is the symbolic Boolean state representing the state space, which is the set of every possible Boolean assignment to 3 signals. Example 3.3.1 uses GV* to reduce the input state space from 2^3 to just 2 states, which are $\{(1, 1, 1), (0, 0, 0)\}$ and symbolically represented by $(\neg(i_0 \wedge i_1), (i_0 \wedge \neg i_1), \neg(i_0 \wedge i_1))$. However, in this case, the reduction in state space comes at a small cost: it uses 3 more AND gates than (v_2, v_1, v_0) . GV**, on the other hand, focuses on reducing the size: $(\neg i_1, \neg i_1, \neg i_0)$ represents the value input state space with 0 AND gates. The value input state space is $\{(1, 1, 1), (1, 1, 0), (0, 0, 1), (0, 0, 0)\}$, which is still smaller than the state space represented by (v_2, v_1, v_0) .

For many designs and properties, we are able to use guard-value encoding to dramatically reduce the input state space and the size of the memories, which results in value verification significantly outperforms BMC. To preserve the soundness of input space reduction and memory abstraction in the value field, we need to:

1. assign values to the guards of the inputs and memory signals appropriately, such that the guard-value pair of the stimulus of these input signals and traces of the memory signals are abstractions of the original ones in $\mathcal{M}$.
2. do guard verification, which verifies that X cannot propagate to the output. In other words, it checks whether there is a trace that is not in $\mathcal{L}(\mathcal{M}_v)$ but abstracted by a trace in $\mathcal{L}(\hat{\mathcal{M}})$ such that it fails the property. The cost of guard verification is the price we have to pay in return of the effectiveness in $\mathcal{M}_v$ which is used in value verification.

Remark 1. *The value verification of BMC-GV accurately checks a small subset of traces of $\mathcal{M}$ using BMC. In other words, the value verification of BMC-GV verifies an under-approximation of $\mathcal{M}$. The guard verification coarsely checks the rest of the traces by verifying whether X at the inputs and the initial state can propagate the output.*

4.4 Other Advantages of the Ternary Verification Based on the Guard-value Encoding

In Chapter 5 and Chapter 6, we show how to leverage the guard-value encoding to achieve data abstraction and memory abstraction, which could be time-consuming tasks. Therefore, it is very important to be able to estimate how much the data abstraction and memory abstraction could benefit the verification. Fortunately, for the guard-value based ternary verification, we can use the performance of the value verification to gauge how well the ternary verification could perform because:

- how much the value verification is faster than the Boolean verification decides the upper bound of how much the ternary verification could improve over the Boolean verification.
- ease of use: as stated in Remarks 1, the value verification is simply the Boolean verification but with reduced state space, which makes it easy to run and test the value verification. In practice, to estimate how much the value verification improves the Boolean verification, we usually sketch how the input value state space is approximately reduced by the data abstraction and what the value transition functions of memory signals approximately become after the memory abstraction without worrying how the guard field is affected.

Another advantage is also associated with Remarks 1. Because the value verification checks an under-approximation of the original design, the bugs it finds are always real bugs while for the ternary verification based on other encodings, additional steps have to be taken to determine whether the bugs found are real or due to too much abstraction. Besides, because the reduced state space of the value verification, it can often find bugs faster than the Boolean verification. In summary, the guard-value encoding based ternary verification is also good at finding bugs in design.

Chapter 5

Data Abstraction

We present a novel data abstraction technique that reduces much more of the state space than conventional ternary abstraction’s approach of over-approximating a set of Boolean values with a smaller set of ternary values. We first illustrate the main idea of our data abstraction with a realistic CAM verification in Section 5.1. Next, we formalize our data abstraction in Section 5.2 and demonstrate how to use it in practice with general examples. We also show how our data abstraction can enable bit-width reduction which helps further simplify verification problems.

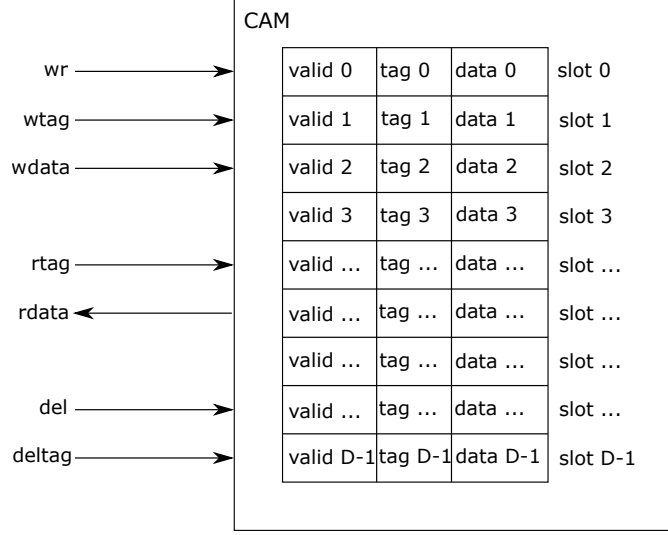
5.1 A Motivating Example

In this section, we verify a realistic hardware design of a content-addressable-memory (CAM). To improve the verification, we incrementally introduce the ternary domain, guard-value encoding and symbolic indexing, which altogether reduce the input state space. We will formalize and expand this idea in Section 5.2 and call it data abstraction with the guard-value encoding.

CAM is parameterized by the number of memory slots D , the width of the tag TW and the width of the data DW . Figure 5.1 shows the CAM’s interface and that in each memory slot, there are tag, data and valid flag, which indicates whether the slot is used. Our CAM has 3 operations and each one takes 1 clock cycle: 1) write **wdata** by asserting the input signal **wr** to the slot whose tag matches the write tag input **wtag**. If **wtag** doesn’t already exist in the CAM, the first unused slot will be filled with **wtag** and **wdata**; 2) read (by de-asserting the input **wr**) from the slot whose tag matches the read tag input **rtag**; and present the data at the output **rdata**; 3) delete (by asserting the input signal **del**) the slots whose tags match the delete tag input **deltag** by clearing the valid flags of the matching slots.

We verify the data correctness property which states that reading based on a tag $\overrightarrow{vtag}$ should return the last data written with that tag, where $\overrightarrow{vtag} = (vtag_{(TW-1)}, \dots, vtag_1, vtag_0)$

Figure 5.1: Content-addressable memory



is a vector of TW Boolean variables from V .

When verifying the data correctness property, we do not care about the exact values of $\mathbf{rtag}$, $\mathbf{wtag}$ and the tags of memory slots. Instead, we only care about whether they are equal to $\overrightarrow{vtag}$. With the ternary domain, we can use X to abstract away the information that is unnecessary to determine whether a value is equal to $\overrightarrow{vtag}$. For example, $(\neg vtag_{(TW-1)}, v_{TW-2}, \dots, v_0)$, where $\vec{v}$ is a vector of TW Boolean variables, is not equal to $\overrightarrow{vtag}$ regardless of the values for $\vec{v}$. Therefore, we can abstract them with X : $(\neg vtag_{(TW-1)}, X, \dots, X)$. This way, we can abstract the set of every possible vector that is not equal to $\overrightarrow{vtag}$ with

$$\{(\neg vtag_{(TW-1)}, X, \dots, X), (X, \neg vtag_{(TW-2)}, X, \dots, X), \dots, (X, \dots, X, \neg vtag_0)\}$$

For each vector from the above set, all but 1 bit are X , which provides great flexibility if we use the guard-value encoding. By using $(0, \neg vtag_i)$ to represent X at the i -th location of a vector, the value component for each vector all becomes $(\neg vtag_{(TW-1)}, \dots, \neg vtag_1, \neg vtag_0)$, which is a bit-wise negation of $\overrightarrow{vtag}$. Therefore, in $\mathcal{M}_v$, which is the Boolean transition system we use for value verification of BMC-GV, the domain for $\mathbf{wtag}$, $\mathbf{rtag}$ and the memory tags is reduced to just 2 symbolic values $\{\overrightarrow{vtag}, \neg \overrightarrow{vtag}\}$, where we use $\neg \overrightarrow{vtag}$ to denote bit-wise negation of $\overrightarrow{vtag}$. In the guard verification of BMC-GV, we also need to know the guard values for these signals. To construct a Boolean function vector such that it generates the set of the guard fields of the vectors, we use symbolic indexing. Firstly, we label vectors from the set by $TW - 1$ downto 0 and create a vector of $\lceil \log(TW) \rceil$ of fresh Boolean variables, $\overrightarrow{id\bar{x}}$, to index among $TW - 1, \dots, 1, 0$. This way, for the vector labeled with i , only its i -th bit is non- X . Therefore, the vector of guards is $(i\bar{x} = TW - 1, \dots, i\bar{x} = 1, i\bar{x} = 0)$ ¹.

¹Here we assume that TW is a power of 2.

Without using ternary domain and guard-value encoding, the domain will include TW symbolic values and there will be TW^2 fresh Boolean variables used (see Formula 5.1). In comparison, $\{\vec{vtag}, \neg\vec{vtag}\}$ has 2 symbolic values and uses TW fresh Boolean variables.

Below shows the set of all symbolic values that are not equal to $\vec{vtag}$. In the formula, $\vec{v}^0, \vec{v}^1, \dots$, and $\vec{v}^{TW-1}$ are different vectors of TW Boolean variables.

$$\begin{aligned} & \{(\neg vtag_{(TW-1)}, v_{TW-2}^{TW-1}, \dots, v_0^{TW-1}), \\ & \quad (\dots, \dots, \dots,) \\ & \quad (v_{(TW-1)}^1, v_{(TW-2)}^1, \dots, \neg vtag_1, v_0^1), \\ & \quad (v_{(TW-1)}^0, v_{(TW-2)}^0, \dots, v_1^0, \neg vtag_0)\} \end{aligned} \quad (5.1)$$

5.2 Data Abstraction with the Guard-value Encoding

When verifying a hardware design, it is common that for some signals, we do not care about their exact values. Instead, we only care about whether their values have the characteristics described by some predicates. In the motivating example, $\vec{wtag}$, $\vec{rtag}$, $\vec{deltag}$ and the tags of the CAM memory slots are vectors of signals and we care about whether their values are equal to $\vec{vtag}$. The corresponding predicate is $p(\vec{n}, \vec{vtag}) \equiv \vec{n} = \vec{vtag}$, where $\vec{n}$ is a vector of $|\vec{vtag}|$ signals. The predicates partition the domain of relevant signals. For example, $p(\vec{n}, \vec{vtag})$ partitions the domain into 2 for any assignment to $\vec{vtag}$, where 1 partition has all values that are not equal to $\vec{vtag}$ while the other has $\vec{vtag}$. For all states belonging to the same partition, every predicate is evaluated to the same value. Thus, the states in a partition are indistinguishable by the predicates. Ternary abstraction uses ternary values to compactly represent each partition. The ternary values representing a partition will also be called ternary representatives. In many cases, the number of ternary representatives of a partition can be exponentially smaller than the number of elements in the partition. For example, there are $2^n - 1$ Boolean states that are not equal to all 1s. But this partition can be represented with just n ternary states

$$\{(0, X, \dots), (X, 0, X, \dots), \dots, (X, \dots, 0)\} \quad (5.2)$$

The guard-value encoding can help further reduce the size of the domain for the value verification using the flexibility in encoding X. Recall that in the value verification, a signal's domain is Boolean instead of ternary. The reduced domain used in the value verification is composed of Boolean representatives which are members of the original domain of the signals. To demonstrate the power of the data abstraction aided by the guard-value encoding, let us use $\langle 0, 0 \rangle$ to encode X in (5.2). Then its value field becomes just $\{(0, \dots, 0)\}$. Our data abstraction is a significant improvement on the existing data abstraction using ternary logic and we will formalize it in the next section. To make our data abstraction practical and useful, we will summarize common predicates used to capture needed information for verification, which include “equal to a symbolic vector”,

“less than a symbolic vector” and “equal to either of the symbolic vectors”. For each predicate, we will show how to apply our data abstraction to exponentially reduce the size of the value domain. For predicates such as “equal to a symbolic vector”, the values in the reduced domain satisfy the conditions that enable bit-width reduction, which helps further simplify the verification problem.

The data abstraction technique in this paper is usually applied to the *input* signals’ state space and the *initial* state space of register signals, which are also called “domain” of these signals. There are 3 use cases because the domain could be a subset of every possible value. But we show that they can all be reduced to the basic first case. Therefore, we will describe our data abstraction assuming the first use case.

1. Case 1: the domain of k signals includes every possible value, i.e., the domain is $\mathbb{B}^k$. We have identified a set of predicates that describes the needed information for verification. In this case, the data abstraction reduces the domain to a union of representatives from each partition for the value verification.
2. Case 2: the domain of k signals is a subset of $\mathbb{B}^k$ characterized by p and p captures all of the information we need of the values from the domain. It can be reduced to case 1 by discarding the partition not satisfying p . Therefore, our data abstraction reduces the domain to a set of representatives that satisfy p .
3. Case 3: similar to Case 2, the domain of k signals is a subset of $\mathbb{B}^k$ characterized by p . However, besides p , we need more predicates to fully capture the information needed for verification. This case can be reduced to the first case by considering all of the predicates including p when partitioning the domain but discarding any subset where p is not satisfied.

5.2.1 Formalization

Let us consider the domain for a set of k signals. The characteristics we care about for the values from the domain are described by m predicates, denoted by $p_0, p_1, \dots, p_{m-1}$. We will first consider the cases where the characteristics are independent of the environment variables, i.e., the inputs to the predicates are just the k signals. Later, we will extend it to the general cases, where the characteristics are dependent on some Boolean variables such as $p(\vec{n}, \overrightarrow{vtag})$.

Definition 13 (Distinguishable values). *Two values q_0 and q_1 from the domain $\mathbb{B}^k$ are distinguishable by the set of predicates, $p_0, p_1, \dots, p_{m-1}$, iff $\exists i < m . p_i(q_0) \neq p_i(q_1)$.*

m predicates partition the domain into 2^m subsets. Each subset comprises of the values that cannot be distinguished by the set of predicates. We define the i -th subset as

$$D_i = \{q : (p_{m-1}(q), \dots, p_1(q), p_0(q)) \text{ is a Boolean string representing the integer } i\}$$

For each subset, using ternary logic, we can abstract it with a set of vectors of ternary values, where we retain just enough information to satisfy of the subset's predicate. Using symbolic indexing, the abstraction set is represented as a vector of symbolic ternary values, which is encoded by a vector of two symbolic Boolean values in guard-value encoding, i.e., the guard and value. By leveraging the flexibility in guard-value encoding, we can achieve different optimization goals. For data abstraction, it is most important to reduce the size of the domain for the value field.

Definition 14 (Data abstraction with the guard-value encoding).

- A vector of k symbolic ternary values, $\overrightarrow{\langle f_g, f_v \rangle} = (\langle f_{g_{k-1}}, f_{v_{k-1}} \rangle, \dots, \langle f_{g_1}, f_{v_1} \rangle, \langle f_{g_0}, f_{v_0} \rangle)$ is a data abstraction of D_i if the following requirements are satisfied:

1. *correctness requirement:* Let $\hat{F}$ be the set of ternary values generated by $\overrightarrow{\langle f_g, f_v \rangle}$, i.e., $\hat{F} = \{\hat{q} : \exists env . \hat{q} = \overrightarrow{\langle f_g(env), f_v(env) \rangle}\}$. The correctness requirement is that 1) for any value from the subset D_i , it is abstracted by a ternary value from $\hat{F}$ and 2) for any ternary value $\hat{q}$ from $\hat{F}$, all Boolean values q' abstracted by it belong to D_i :

$$(p_{m-1}(q'), \dots, p_1(q), p_0(q)) = i.$$

2. *effectiveness requirement:* the domain of the value field of $\overrightarrow{\langle f_g, f_v \rangle}$, i.e., the domain for $\overrightarrow{f_v} = (f_{v_{k-1}}, \dots, f_{v_1}, f_{v_0})$, must be smaller than the size of D_i :

$$|\{q : \exists env . q = \overrightarrow{f_v}(env)\}| < |D_i|$$

In fact, the smaller $|\{q : \exists env . q = \overrightarrow{f_v}(env)\}|$ is, the more effective the data abstraction becomes.

- For a domain $\mathbb{B}^k$ partitioned by m predicates, the data abstraction with the guard-value encoding is achieved by representing each one of the 2^m partition with a vector of k symbolic ternary values that it is a data abstraction of the partition.

Remark 2.

- Let we construct $\hat{p}_{m-1}, \dots, \hat{p}_1$ and $\hat{p}_0$ from $p_{m-1}, \dots, p_1$ and p_0 respectively by replacing the Boolean operators with ternary ones. Because $\hat{p}_{m-1}, \dots, \hat{p}_1$ and $\hat{p}_0$ are monotonic with regard to the abstraction relation based on Proposition 1, a sufficient condition for the second requirement of the correctness is:

$$(\hat{p}_{m-1}(\hat{q}), \dots, \hat{p}_1(\hat{q}), \hat{p}_0(\hat{q})) = i.$$

- Satisfying the correctness requirement implies that D_i is the maximum representable set (MRS) of $\overrightarrow{\langle f_g, f_v \rangle}$ (Lemma 7).

Lemma 7. If $\overrightarrow{\langle f_g, f_v \rangle} = (\langle f_{g_{k-1}}, f_{v_{k-1}} \rangle, \dots, \langle f_{g_1}, f_{v_1} \rangle, \langle f_{g_0}, f_{v_0} \rangle)$ is a data abstraction of the i -th subset (D_i) of $\mathbb{B}^k$, then D_i is the MRS of $\overrightarrow{\langle f_g, f_v \rangle}$.

Proof. Recall that a subset of Boolean values is the MRS of a symbolic ternary value iff 1) for any value from the subset, it can be abstracted by a ternary value generated by the symbolic ternary value and 2) for any Boolean value that can be abstracted by a ternary value generated by the symbolic ternary value, it must also be in the subset. The first requirement is trivially satisfied for D_i by definition. For the second requirement, let us assume that there is a Boolean value q that is not in D_i , but it can be abstracted by a ternary value $\hat{q}$ generated by $\overrightarrow{\langle f_g, f_v \rangle}$. Then according to the definition:

$$(\hat{p}_{m-1}(\hat{q}), \dots, \hat{p}_1(\hat{q}), \hat{p}_0(\hat{q})) = i.$$

Because ternary interpretations of Boolean functions are monotonic and they are as if Boolean functions when their inputs are Boolean, we have

$$(p_{m-1}(q), \dots, p_1(q), p_0(q)) = i.$$

q must belong to D_i , which contradicts our assumption. Therefore, the second requirement is also satisfied. $\square$

Because of the inherent advantage of ternary domain to represent a set of Boolean values compactly and the flexibility in guard-value encoding, in practice, we can often significantly reduce the size of the domain for the value field of some signals, which will dramatically accelerate the value verification.

Example 5.2.1. The predicate $f(n_2, n_1, n_0) \equiv (n_2, n_1, n_0) = (1, 0, 1)$ partitions the domain for (n_2, n_1, n_0) into 2, which are $\{(1, 0, 1)\}$ and $\{(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 1, 0), (1, 1, 1)\}$. The first partition has just one element and in our data abstraction it is trivially abstracted by itself. The guard-value encoding encodes it with $(\langle 1, 1 \rangle, \langle 1, 0 \rangle, \langle 1, 1 \rangle)$. The other partition is abstracted with $\{(0, \mathbf{X}, \mathbf{X}), (\mathbf{X}, 0, \mathbf{X}), (\mathbf{X}, \mathbf{X}, 0)\}$. By using $\langle 0, 0 \rangle$ to encode $\mathbf{X}$, the partition's guard value encoding is

$$\begin{aligned} & \{(\langle 1, 0 \rangle, \langle 0, 0 \rangle, \langle 0, 0 \rangle), \\ & (\langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle 0, 0 \rangle), \\ & (\langle 0, 0 \rangle, \langle 0, 0 \rangle, \langle 1, 0 \rangle)\} \end{aligned}$$

It is easy to see that for any ternary value from the partition, its value field is always $(0, 0, 0)$, which is key to make our data abstraction effective (see Definition 14). With symbolic indexing, the partition can be represented by using 2 Boolean variables: $(\langle v_1, 0 \rangle, \langle \neg v_1 \wedge v_0, 0 \rangle, \langle \neg v_1 \wedge \neg v_0, 0 \rangle)$. It takes 3 Boolean variables to represent the domain $\mathbb{B}^k$ for (n_2, n_1, n_0) . This example is extended in Example 5.2.2 and generalized for any number of signals in Section 5.2.2, where we will show the reduction in the number of Boolean variables needed to symbolic represent the domain is exponential.

In practice, the partition of a domain is often dependent on some Boolean variables, as shown in the motivating example (Section 5.1). We use the vector $\vec{c}$ to denote these Boolean variables. In this case, the inputs to the characteristic predicates not only include signals but also variables from $\vec{c}$. Each assignment to $\vec{c}$ determines a partition, which causes that a concrete value may belong to different subsets for different assignments to $\vec{c}$. Therefore, in order to represent the values that are in the same subset regardless of the assignment to $\vec{c}$, we extend the domain $\mathbb{B}^k$ to include vectors of k Boolean expressions in terms of the variables from $\vec{c}$. The extended domain is denoted by $\mathbb{B}^{+k} = \{(e_{k-1}, \dots, e_1, e_0) : e_i \text{ is a Boolean expression in terms of } \vec{c}\}$.

Definition 15 (Distinguishable symbolic values). *Two symbolic values q_0 and q_1 from the domain $\mathbb{B}^{+k}$ are distinguishable by the set of predicates, $p_0, p_1, \dots, p_{n-1}$, iff $\exists i < n . \forall env . p_i(q_0(env), env) \neq p_i(q_1(env), env)$, where env represents an assignment to $\vec{c}$ and recall that $\vec{c}$ are also among the inputs of p_i .*

The key of extending the definition of data abstraction to accommodate predicates with Boolean variables is to realize that there are now two types of Boolean variables used in the vector of symbolic ternary values abstracting a subset. The two types of variables are the ones used in the predicates (i.e., $\vec{c}$) and the variables created by symbolic indexing. We denote the symbolic indexing variables by $\vec{v}$. Accordingly, env_c is an assignment to $\vec{c}$ and env_v is an assignment to $\vec{v}$. Given env_v , a vector of symbolic ternary values evaluates to a set of vectors of symbolic ternary values in terms of variables from $\vec{c}$.

Definition 16 (Parameterized Data abstraction with the guard-value encoding). *A vector of k symbolic ternary values, denoted by $\langle \vec{f}_g, \vec{f}_v \rangle = (\langle f_{g_{k-1}}, f_{v_{k-1}} \rangle, \dots, \langle f_{g_1}, f_{v_1} \rangle, \langle f_{g_0}, f_{v_0} \rangle)$, is a parameterized data abstraction of a domain that is a subset of $\mathbb{B}^k$, characterized by the predicate $p_1(\vec{n}, \vec{c})$, iff for any assignment to $\vec{c}$, denoted by env_c , $\langle \vec{f}_g, \vec{f}_v \rangle(env_c)$ is a data abstraction of the domain.*

The parameterized data abstraction for the domain $\mathbb{B}^k$ partitioned by m characteristic predicates $p_0(\vec{n}, \vec{c}), p_1(\vec{n}, \vec{c}), \dots, p_{m-1}(\vec{n}, \vec{c})$ is achieved iff for any env_c , and for each partitioned subset evaluated with env_c , its corresponding vector of symbolic values after evaluation with env_c is a data abstraction of the subset.

Example 5.2.2. *This example builds on Example 5.2.1 by comparing to a vector of variables rather than the concrete vector $(1, 0, 1)$. The predicate $f(n_2, n_1, n_0, c_2, c_1, c_0) \equiv (n_2, n_1, n_0) = (c_2, c_1, c_0)$ partitions the domain for (n_2, n_1, n_0) into 2 subsets, which are $\{(q_2, q_1, q_0) : (q_2, q_1, q_0) = (c_2, c_1, c_0)\}$ and $\{(q_2, q_1, q_0) : (q_2, q_1, q_0) \neq (c_2, c_1, c_0)\}$. When $(c_2, c_1, c_0) = (1, 0, 1)$, this example is reduced to Example 5.2.1. Data abstraction with the guard-value encoding abstracts the first subset with $\{(c_2, c_1, c_0)\}$, which is encoded by $(\langle 1, c_2 \rangle, \langle 1, c_1 \rangle, \langle 1, c_0 \rangle)$ using guard-value encoding. The 2nd subset is abstracted with $\{(\neg c_2, X, X), (X, \neg c_1, X), (X, X, \neg c_0)\}$, which is represented by the symbolic guard-value pair $(\langle v_1, \neg c_2 \rangle, \langle \neg v_1 \wedge v_0, \neg c_1 \rangle, \langle \neg v_1 \wedge \neg v_0, \neg c_0 \rangle)$ using symbolic indexing with the goal of optimizing the size of the domain for the value field. The domain for the value field of (n_2, n_1, n_0)*

is just $\{(\neg c_2, \neg c_1, \neg c_0)\}$, thus achieving a significant reduction from the second subset. The domain for the guard field of (n_2, n_1, n_0) is $\{(v_1, \neg v_1 \wedge v_0, \neg v_1 \wedge \neg v_0)\}$.

It is important to note that our data abstraction is not suitable for all hardware verification problems. Especially when there are too many predicates and/or each predicate is complex, the effectiveness requirement would become difficult to achieve. For this reason, in general, arithmetic designs are not a good candidate to apply our data abstraction.

Nonetheless, there are plenty of hardware designs where our data abstraction is very effective. In Section 5.2.2 - 5.2.4, we demonstrate the idea of data abstraction with the guard-value encoding by manually implementing the data abstraction for a selection of predicates that are very useful to partition the domain for a range of hardware designs and properties to verify.

5.2.2 Equal to a Vector of Boolean Variables

Just like verifying properties of a CAM in Section 5.1, for many cases, it is sufficient to know whether a vector of signals is equal to a vector of Boolean variables. Let us assume that the width of the vector is k and the domain is $\mathbb{B}^k$. The predicate is $eq(\vec{n}, \vec{c}) \equiv \vec{n} = \vec{c}$, where $\vec{c}$ is a vector of k Boolean variables and $\vec{n}$ is a vector of k signals. The predicate divides $\mathbb{B}^k$ into two subsets. The first subset's size is 1 and it includes the value that is equal to $\vec{c}$ while the other subset's size is $2^k - 1$ and it includes values that are not equal to $\vec{c}$. The exact values in either of the subset obviously depend on the assignment to $\vec{c}$.

We have used this predicate to partition a domain in the motivation example and Example 5.2.2. We will first generalize it for any width of the vector. When there are spurious CEXs, a common reason is due to the unknown results (i.e., X) when comparing two values that neither is equal to the vector of Boolean variables: if $\vec{a} \neq \vec{c}$ and $\vec{b} \neq \vec{c}$, we do not know whether $\vec{a} = \vec{b}$. Around the end of Section 7.5 shows how to eliminate such spurious CEXs. In this section, we will also show how our data abstraction enables reducing the bit-width of vectors.

Data Abstraction

With data abstraction, the partitions for $eq(\vec{n}, \vec{c})$ are generated by the following 2 symbolic ternary vectors respectively, where $\vec{v}$ is a vector of $\lceil \log(k) \rceil$ Boolean variables needed by symbolic indexing.

$$\begin{aligned} &= \vec{c} : (\langle 1, c_{k-1} \rangle, \quad \langle 1, c_{k-2} \rangle, \quad \dots, \langle 1, c_0 \rangle) \text{ and} \\ &\neq \vec{c} : (\langle \vec{v} \geq k-1, \neg c_{k-1} \rangle, \langle \vec{v} = k-2, \neg c_{k-2} \rangle, \dots, \langle \vec{v} = 0, \neg c_0 \rangle) \end{aligned} \quad (5.3)$$

The guard for the partition $\neq \vec{c}$ selects a signal from $\vec{n}$ whose value is different from the corresponding bit of $\vec{c}$. Because the range of $\vec{v}$ is from 0 to $2^{\lceil \log(k) \rceil}$ which is larger than k

when k is not a power of 2, we make $k \geq k-1$ index the signal at the highest location, i.e., $n_{(k-1)}$. The value field is $\vec{c}$ for the first subset and $\neg\vec{c}$ for the second subset. Therefore, the domain for the value field is just $\{\vec{c}, \neg\vec{c}\}$. Recall that $\neg\vec{c}$ is a bit-wise negation of $\vec{c}$. For any env_c , the size of the domain for value field is 2, which is exponentially smaller than the size of $\mathbb{B}^k$.

Lemma 8. *Formula (5.3) is a data abstraction for $\mathbb{B}^k$ given the predicate $eq(\vec{n}, \vec{c}) \equiv \vec{n} = \vec{c}$*

Proof. Formula (5.3) satisfies the effectiveness requirement because it reduces the domain size exponentially. Let us show that it also satisfies the correctness requirement. We specifically look at the abstraction for the second subset because the first one is straightforward.

According to Formula (5.3), for any 2 signals from $\vec{n}$, their guards can not be 1 at the same time and for any assignment env_v , one of the signals' guards must be 1. Therefore, by enumerating env_v , the generated ternary vectors are:

$$\begin{aligned} & (\langle 1, \neg c_{k-1} \rangle, \langle 0, \neg c_{k-2} \rangle, \dots, \langle 0, \neg c_0 \rangle) \\ & (\langle 0, \neg c_{k-1} \rangle, \langle 1, \neg c_{k-2} \rangle, \dots, \langle 0, \neg c_0 \rangle) \\ & \dots \\ & (\langle 0, \neg c_{k-1} \rangle, \langle 0, \neg c_{k-2} \rangle, \dots, \langle 1, \neg c_0 \rangle) \end{aligned}$$

Encoding the vectors of ternary values:

$$\begin{aligned} \mathbb{T}_{\vec{n}} = \{ & (\neg c_{k-1}, X, \dots, X), \\ & (X, \neg c_{k-2}, X, \dots, X), \\ & \dots, \\ & (X, X, \dots, \neg c_0) \} \end{aligned}$$

For any value $q \in \mathbb{B}^k$ that is not equal to $\vec{c}$, at least 1 bit is different. Let it be the i -th bit, then q is abstracted by $(\dots, X, \neg c_i, X, \dots)$ from $\mathbb{T}_{\vec{n}}$. Besides, it is easy to verify that for any ternary value $\hat{q} \in \mathbb{T}_{\vec{n}}$, $\hat{eq}(\hat{q}, \vec{c}) = 0$ for any assignment to $\vec{c}$. As a conclusion, the correctness requirement is satisfied. $\square$

When verifying hardware designs, it is often the case that we verify a property for all possible requests, responses, processes, memory accesses and resources by verifying for an arbitrary one that is identified by a symbolic “identifier”. The identifier could be implicit in the design. For example, a memory access can be identified by the address of the access. Let the symbolic identifier be $\vec{c}$. Then $eq(\vec{n}, \vec{c}) \equiv \vec{n} = \vec{c}$ is the predicate we use to partition the domain of the signals relevant to the IDs.

Bit-width Reduction

Let $\vec{a}$ and $\vec{b}$ be 2 vectors of signals whose domain in value verification is reduced to $\{\vec{c}, \neg\vec{c}\}$ using data abstraction with the predicate $eq(\vec{n}, \vec{c}) = \vec{n} \neq \vec{c}$. Then, $\vec{a} = \vec{b}$ and $\vec{a} \neq \vec{b}$ can be simplified to $a_0 = b_0$ and $a_0 \neq b_0$ respectively, because of Lemma 9.

Lemma 9. $\forall i < |\vec{a}| . \vec{a} = \vec{b} \text{ iff } a_i = b_i$. *Equivalently, $\forall i < |\vec{a}| . \vec{a} \neq \vec{b} \text{ iff } a_i \neq b_i$.*

Proof. For any two values from the domain $\{\vec{c}, \neg\vec{c}\}$, they are either equal to each other for every bit or not equal to each other for every bit. Therefore, we can decide their equality relationship by just looking at any bit. $\square$

In practice, for signals whose domain are reduced to $\{\vec{c}, \neg\vec{c}\}$ for value verification, they are usually only used as operands for the operators $\{=, \neq\}$. Therefore, we are able to replace every occurrence of the signals with the 0-th bit, i.e., reducing the bit-width of these signals to just 1 bit. We demonstrate its effectiveness in Figure 8.8 and tables such as 8.4.

5.2.3 Equal to Either of Two Vectors of Boolean Variables

In this section, we show how to construct data abstraction when it is not sufficient to only know whether a vector of signals is equal to a vector of Boolean variables, but we also need to know its equality relationship with another vector of Boolean variables. For example, our CAM (Section 5.1) supports delete and write at the same cycle, we need to know the equality relationship between the tags of the existing CAM entries with both of the delete tag and write tag. Consider the case where we only know that the tags of the existing CAM entries are not equal to the write tag, but do not know whether they are equal to the delete tag. In the ternary verification, this implies that they could all be equal to the delete tag, thus all getting deleted and updated with the same write tags at the same cycle, which is a spurious CEX to the uniqueness property.

The corresponding predicate we use to partition a domain $(\mathbb{B}^k)$ is $eq_2(\vec{n}, \vec{c}, \vec{d}) = (\vec{n} = \vec{c}) \vee (\vec{n} = \vec{d})$, where $\vec{c}$ and $\vec{d}$ are vectors of k Boolean variables and $\vec{n}$ is a vector of k signals. The predicate partitions $\mathbb{B}^k$ into two subsets. The first subset is $\{\vec{c}, \vec{d}\}$ and when $\vec{c} = \vec{d}$, it is reduced to $\{\vec{c}\}$. Accordingly, the other subset's size is either $2^k - 1$ or $2^k - 2$, and it includes every value that is not equal to either of $\vec{c}$ or $\vec{d}$. The exact values in either of the subsets obviously depend on the assignment to $\vec{c}$ and $\vec{d}$.

The first subset is straightforwardly generated by the following symbolic ternary vector:

$$(\langle 1, v_0 ? c_{k-1} : d_{k-1} \rangle, \langle 1, v_0 ? c_{k-2} : d_{k-2} \rangle, \dots, \langle 1, v_0 ? c_0 : d_0 \rangle) \quad (5.4)$$

sel is an indexing variable that is introduced to arbitrarily select $\vec{c}$ or $\vec{d}$ whose guard value encodings are $(\langle 1, c_{k-1} \rangle, \langle 1, c_{k-2} \rangle, \dots, \langle 1, c_0 \rangle)$ and $(\langle 1, d_{k-1} \rangle, \langle 1, d_{k-2} \rangle, \dots, \langle 1, d_0 \rangle)$.

As for the other subset, it is an intersection of

$$\begin{aligned} D_{\vec{n}}^0 &= \{q(\vec{n}) : q(\vec{n}) \neq \vec{c}\} \\ D_{\vec{n}}^1 &= \{q(\vec{n}) : q(\vec{n}) \neq \vec{d}\}, \end{aligned}$$

where $q(\vec{n})$ denotes an assignment to the signals in $\vec{n}$. Recall that as shown in the previous section, $D_{\vec{n}}^0$ and $D_{\vec{n}}^1$ are abstracted with two sets of ternary values respectively, which are

$$\begin{aligned} \mathbb{T}_{\vec{n}}^0 &= \{(\neg c_{k-1}, X, \dots, X), (X, \neg c_{k-2}, X, \dots, X), \dots, (X, X, \dots, \neg c_0)\} \text{ and} \\ \mathbb{T}_{\vec{n}}^1 &= \{(\neg d_{k-1}, X, \dots, X), (X, \neg d_{k-2}, X, \dots, X), \dots, (X, X, \dots, \neg d_0)\} \end{aligned}$$

To compute a set of ternary values abstracting the second subset, i.e., $D_{\vec{n}}^0 \cap D_{\vec{n}}^1$, we apply intersection on $\mathbb{T}_{\vec{n}}^0$ and $\mathbb{T}_{\vec{n}}^1$. Because a ternary value abstracts a set of Boolean values, we can apply intersection on two ternary values, which returns the ternary value abstracting the intersection of the sets of Boolean values, that are abstracted by the two ternary values (Definition 17). For example,

$$(\neg c_{k-1}, X, \dots, X) \cap (X, \neg d_{k-1}, X, \dots, X) = (\neg c_{k-1}, \neg d_{k-1}, X, \dots, X)$$

Definition 17 (Intersection of 2 ternary vectors). *Let $\hat{a}$ and $\hat{b}$ be two ternary vectors of the same length k . The intersection of $\hat{a}$ and $\hat{b}$, denoted by $\hat{o} = \hat{a} \cap \hat{b}$, can be computed bit-wise:*

$$\forall i < k . \hat{o}_i = \begin{cases} \hat{a}_i, & \text{if } \hat{b}_i = X \text{ or } \hat{b}_i = \hat{a}_i \\ \hat{b}_i, & \text{if } \hat{a}_i = X \\ \perp, & \text{otherwise, i.e., if } \hat{a}_i \neq X, \hat{b}_i \neq X \text{ and } \hat{a}_i \neq \hat{b}_i \end{cases}$$

$\hat{o}$ is “empty” if at least one of its bits is $\perp$.

Let $f_j^i = c_i = d_i ? X : \neg c_j$, then the result of $\mathbb{T}_{\vec{n}}^0 \cap \mathbb{T}_{\vec{n}}^1$ is k^2 ternary vectors. We put them into a $k \times k$ grid.

	$k-1$	$k-2$	$\dots$	0
$k-1$	$\{(\neg d_{k-1}, f_{k-2}^{k-1}, \dots, f_0^{k-1}), (\neg c_{k-1}, \neg d_{k-2}, X, \dots, X), \dots, (\neg c_{k-1}, X, \dots, X, \neg d_0)\}$			
$k-2$	$(\neg d_{k-1}, \neg c_{k-2}, X, \dots, X), (f_{k-1}^{k-2}, \neg d_{k-2}, f_{k-3}^{k-2}, \dots, f_0^{k-1}), \dots, (X, \neg c_{k-1}, \dots, X, \neg d_0)$			
$\dots$	$\dots$			
0	$(\neg d_{k-1}, X, \dots, X, \neg c_0), (X, \neg d_{k-2}, X, \dots, X, \neg c_0), \dots, (f_{k-1}^0, \dots, f_1^0, \neg d_0)\}$			

We label each value from the set with $(row, column)$ coordinates in preparation to show how to construct a symbolic ternary vector to represent the set with symbolic indexing. We create two vectors of indexing variables $\overrightarrow{col}$ and $\overrightarrow{row}$ to index a column and row respectively. The vector at the location (i, i) for all $i \leq k-1$ is the result of $(\dots, X, \neg c_i, X, \dots) \cap (\dots, X$

, $\neg d_i, X, \dots$). Depending on whether $c_i = d_i$, the result is either $(X, \neg c_i, X, \dots, X)$ or empty. Because the result is the union of $k \times k$ symbolic ternary vectors, it is correct to replace empty with any of the $k \times k$ symbolic ternary vectors that is not empty. To make it easier for applying symbolic indexing, as shown above, we use $(\neg c_{k-1}, \neg c_{k-2}, \dots, \neg d_i, \neg c_{i-1}, \dots, \neg c_0)$ instead of empty. It is trivial to see that this vector is not equal to either of $\vec{c}$ and $\vec{d}$, i.e., belong to the set $\mathbb{T}_{\mathbf{r}}^0 \cap \mathbb{T}_{\mathbf{r}}^1$. Therefore, it is ok to include it in the set. For any other vector, whose column and row coordinates are not equal, its i -th bit is $\neg c_i$ if it is in the i -th row, $\neg d_i$ if in the i -th column, and X otherwise. Therefore, the guard of the i -th bit of the symbolic ternary vector should $\overrightarrow{row} = i \vee \overrightarrow{col} = i \vee (c_{row} \neq d_{col} \wedge col = row)$.

In order to reduce the domain size for the value field, we use $\langle 0, \neg c_i \rangle$ to encode X at the i -th bit. This way, the value of the i -th bit in a ternary vector is $\neg d_i$ iff $\overrightarrow{col} = i$. Otherwise, it is $\neg c_i$. In summary, the symbolic ternary vector we use to abstract the set of values that are not equal to either of $\vec{c}$ and $\vec{d}$ is:

$$\begin{aligned} & (\langle (\overrightarrow{col} = k - 1 \vee \overrightarrow{row} = k - 1 \vee (c_{\overrightarrow{row}} \neq d_{\overrightarrow{col}} \wedge \overrightarrow{col} = \overrightarrow{row})), (\overrightarrow{col} = k - 1 ? \neg d_{k-1} : \neg c_{k-1}) \rangle, \\ & \langle (\overrightarrow{col} = k - 2 \vee \overrightarrow{row} = k - 2 \vee (c_{\overrightarrow{row}} \neq d_{\overrightarrow{col}} \wedge \overrightarrow{col} = \overrightarrow{row})), (\overrightarrow{col} = k - 2 ? \neg d_{k-2} : \neg c_{k-2}) \rangle, \\ & \dots, \\ & \langle (\overrightarrow{col} = 0 \vee \overrightarrow{row} = 0 \vee (c_{\overrightarrow{row}} \neq d_{\overrightarrow{col}} \wedge \overrightarrow{col} = \overrightarrow{row})), (\overrightarrow{col} = 0 ? \neg d_0 : \neg c_0) \rangle) \end{aligned} \quad (5.5)$$

Example 5.2.3. *The set of ternary vectors abstracting the Boolean vectors that are not equal to either of $(0, 0, 1, 1)$ and $(1, 0, 1, 0)$ is the result of*

$$\begin{aligned} & \{(1, X, X, X), (X, 1, X, X), (X, X, 0, X), (X, X, X, 0)\} \cap \\ & \{(0, X, X, X), (X, 1, X, X), (X, X, 0, X), (X, X, X, 1)\} \end{aligned}$$

It is:

	3	2	1	0
3	{(0, 1, 0, 0),	(1, 1, X, X),	(1, X, 0, X),	(1, X, X, 1)
2	(0, 1, X, X),	(X, 1, X, X),	(X, 1, 0, X),	(X, 1, X, 1)
1	(0, X, 0, X),	(X, 1, 0, X),	(X, X, 0, X),	(X, X, 0, 1)
0	(0, X, X, 0),	(X, 1, X, 0),	(X, X, 0, 0),	(1, 1, 0, 1)}

Each $\overrightarrow{row}$ and $\overrightarrow{col}$ we need has 2 variables. The symbolic ternary vector is:

$$\begin{aligned} & (\overrightarrow{col} = 3 ? 0 : (\overrightarrow{row} = 3 \vee (c_{\overrightarrow{row}} \neq d_{\overrightarrow{col}} \wedge \overrightarrow{col} = \overrightarrow{row}) ? 1 : X) , \\ & \overrightarrow{col} = 2 ? 1 : (\overrightarrow{row} = 2 \vee (c_{\overrightarrow{row}} \neq d_{\overrightarrow{col}} \wedge \overrightarrow{col} = \overrightarrow{row}) ? 1 : X) , \\ & \overrightarrow{col} = 1 ? 0 : (\overrightarrow{row} = 1 \vee (c_{\overrightarrow{row}} \neq d_{\overrightarrow{col}} \wedge \overrightarrow{col} = \overrightarrow{row}) ? 0 : X) , \\ & \overrightarrow{col} = 0 ? 1 : (\overrightarrow{row} = 0 \vee (c_{\overrightarrow{row}} \neq d_{\overrightarrow{col}} \wedge \overrightarrow{col} = \overrightarrow{row}) ? 0 : X)) \end{aligned}$$

With the guard value encoding and using $\langle 0, \neg c_i \rangle$ to encode X at the i -th bit, the symbolic ternary vector is encoded by:

$$\begin{aligned} & \left(\langle \overrightarrow{col} = 3 \vee \overrightarrow{row} = 3 \vee (c_{\overrightarrow{row}} \neq d_{\overrightarrow{col}} \wedge \overrightarrow{col} = \overrightarrow{row}), \overrightarrow{col} = 3 ? 0 : 1 \rangle, \right. \\ & \left. \langle \overrightarrow{col} = 2 \vee \overrightarrow{row} = 2 \vee (c_{\overrightarrow{row}} \neq d_{\overrightarrow{col}} \wedge \overrightarrow{col} = \overrightarrow{row}), \overrightarrow{col} = 2 ? 1 : 1 \rangle, \right. \\ & \left. \langle \overrightarrow{col} = 1 \vee \overrightarrow{row} = 1 \vee (c_{\overrightarrow{row}} \neq d_{\overrightarrow{col}} \wedge \overrightarrow{col} = \overrightarrow{row}), \overrightarrow{col} = 1 ? 0 : 0 \rangle, \right. \\ & \left. \langle \overrightarrow{col} = 0 \vee \overrightarrow{row} = 0 \vee (c_{\overrightarrow{row}} \neq d_{\overrightarrow{col}} \wedge \overrightarrow{col} = \overrightarrow{row}), \overrightarrow{col} = 0 ? 1 : 0 \rangle \right) \end{aligned}$$

The domain for the value field is:

$$\{(0, 1, 0, 0), (1, 1, 0, 0), (1, 1, 0, 1)\}$$

Based on the symbolic ternary vector shown in Formula (5.5), the value field of the generated symbolic ternary vectors in terms of $\vec{c}$ and $\vec{d}$, after enumerating assignments to col and row is

$$\begin{aligned} & \{(\neg d_{k-1}, \neg c_{k-2}, \dots, \neg c_0), \\ & (\neg c_{k-1}, \neg d_{k-2}, \neg c_{k-3}, \dots, \neg c_0), \\ & \dots, \\ & (\neg c_{k-1}, \neg c_{k-2}, \dots, \neg c_1, \neg d_0)\} \end{aligned}$$

There are in total at most k vectors in the set, which is an exponential reduction from the subset's original size which is $2^k - 1$ or $2^k - 2$.

Lemma 10. Formula (5.4) and (5.5) together implement a data abstraction for $\mathbb{B}^k$ given the predicate $eq_2(\vec{n}, \vec{c}, \vec{d}) \equiv (\vec{n} = \vec{c}) \vee (\vec{n} = \vec{d})$

Proof. The proof is similar to that of Lemma 7. □

5.2.4 Less Than a Vector of Boolean Variables

Another common relationship we can leverage for data abstraction when verifying a design is the comparison between vectors of signals in terms of which is larger. For example, an age-based scheduler (details in Section 8.1.2) selects the oldest request to execute. Therefore, we only care about the ' $<$ ' relationship among the ages of the requests, instead their exact values. We will use the predicate $lt(\vec{n}, \vec{c}) = \vec{n} < \vec{c}$ to describe this relationship.

It is again assumed that the domain to partition is $\mathbb{B}^k$. The partitions by the predicate are characterized by $\vec{n} < \vec{c}$ and $\vec{n} \geq \vec{c}$ and whose size are $\vec{c}$ and $2^k - \vec{c}$ respectively. To construct the data abstraction for each partition, we first abstract each with a set of ternary values, and use appropriate value bit to encode each X , with the goal to best satisfy the effectiveness requirement of the data abstraction.

Example 5.2.4. $\{(0, 0, 0), (0, 0, 1), (0, 1, 0)\}$ characterized by $lt(\vec{n}) = \vec{n} < (0, 1, 1)$, can be abstracted by both of $\{(0, 0, X), (0, X, 0)\}$ and $\{(0, 0, X), (0, 1, 0)\}$. Both have the same size. However, $\{(0, 0, X), (0, X, 0)\}$ is preferred because after using $\langle 0, 0 \rangle$ to encode X , the set is encoded by

$$\begin{aligned} & \{(\langle 1, 0 \rangle, \langle 1, 0 \rangle, \langle 0, 0 \rangle) \\ & (\langle 1, 0 \rangle, \langle 0, 0 \rangle, \langle 1, 0 \rangle)\} \end{aligned}$$

Therefore, the value field of the set has just one vector which is $\{(0, 0, 0)\}$. Note that for the other abstract ternary set, no matter how we encode X , the value field always has 2 vectors.

We abstract $\{q(\vec{n}) : q(\vec{n}) < \vec{c}\}$ ² with the set of symbolic ternary vectors in terms of $\vec{c}$ shown in Table 5.1. Its correctness is proven in Lemma 11. There are k ternary vectors in the set and we label them from $k - 1$ down to 0. In the i -th vector, its i -th bit is 0. If c_i is 1, then all bits that are lower than i are X . Otherwise, they are all 0s. The upper bits are X if the corresponding bits in $\vec{c}$ are 1 and c_i is 1. They are 0s otherwise. Therefore, when c_i is 1, the i -th vector abstracts the set of Boolean vectors that are smaller than $\vec{c}$ due to their i -th bits being 0 and the upper bits are either smaller than or equal to the corresponding bits of $\vec{c}$. When c_i is 0, the i -th vector is all 0s.

²Assume that $\vec{c}$ is non-zero, otherwise, it is an empty set.

Table 5.1: Ternary abstraction for the less-than predicate.

label							
$k-1$	$\{(0,$	$(c_{k-1} ? X : 0),$	$\dots,$	$\dots,$	$\dots,$	$(c_{k-1} ? X : 0),$	$\dots,$
$k-2$	$((c_{k-1} \wedge c_{k-2} ? X : 0),$	$0,$	$(c_{k-2} ? X : 0),$	$\dots,$	$\dots,$	$(c_{k-2} ? X : 0),$	$\dots,$
$k-3$	$((c_{k-1} \wedge c_{k-3} ? X : 0),$	$(c_{k-2} \wedge c_{k-3} ? X : 0),$	$0,$	$(c_{k-3} ? X : 0),$	$\dots,$	$(c_{k-3} ? X : 0),$	$\dots,$
$\dots$							
0	$((c_{k-1} \wedge c_0 ? X : 0),$	$(c_{k-2} \wedge c_0 ? X : 0),$	$\dots,$	$\dots,$	$(c_1 \wedge c_0 ? X : 0),$	$0\}$	

Table 5.2: Ternary abstraction for the greater-than or equal-to predicate.

label							
k	$\{(c_{k-1},$	$c_{k-2},$	$\dots,$	$\dots,$	$\dots,$	$c_0),$	
$k-1$	$(1,$	$(\neg c_{k-1} ? X : 1),$	$\dots,$	$\dots,$	$\dots,$	$(\neg c_{k-1} ? X : 1),$	
$k-2$	$((\neg c_{k-1} ? X : 1),$	$1,$	$(\neg c_{k-2} ? X : 1),$	$\dots,$	$\dots,$	$(\neg c_{k-2} ? X : 1),$	
$k-3$	$((\neg c_{k-1} ? X : 1),$	$(\neg c_{k-2} ? X : 1),$	$1,$	$(\neg c_{k-3} ? X : 1),$	$\dots,$	$(\neg c_{k-3} ? X : 1),$	
$\dots$							
0	$((\neg c_{k-1} ? X : 1),$	$(\neg c_{k-2} ? X : 1),$	$\dots,$	$\dots,$	$(\neg c_1 ? X : 1),$	$1\}$	

In order to use symbolic indexing to represent the above set with a single symbolic ternary vector, we create a vector of Boolean variables $\vec{i}$ to index from 0 to $k - 1$. Based on the set shown in Table 5.1, we observe that for the vector labeled with i , its i -th bit is always 0 and for the higher bits, they are 0 iff the corresponding bit in $\vec{c}$ is 0 or c_i is 0. As for the lower bits, they are 0 iff c_i is 0. Let us denote the symbolic ternary vector by $\hat{r}^<$. Then, its guard field can be calculated as below, where $\vec{i}$ is a vector of Boolean variables created to index from $k - 1$ to 0:

$$\forall j < k . \hat{r}_{gj}^< \equiv (\vec{i} = j) \vee ((j > \vec{i}) \wedge (\neg c_j \vee \neg c_{\vec{i}})) \vee ((j < \vec{i}) \wedge \neg c_{\vec{i}})$$

To reduce the size of the domain for the value field, here we use $\langle 0, 0 \rangle$ to encode $\mathbf{X}$. This way, the value field of every vector from the set is $(0, \dots, 0)$. Therefore, $\hat{r}_v^< = (0, \dots, 0)$.

We proceed to construct the data abstraction for the other partition, i.e., $\{q(\vec{n}) : \vec{n} \geq \vec{c}\}$. The idea is very similar. We will skip some steps. We use the following set of ternary vectors shown in Table 5.2 to abstract this partition.

Let us denote the symbolic ternary vector representing this set of ternary vectors by $\hat{r}^{\geq}$. sel is a Boolean variable created to symbolically represent the union of the vector labeled with k , i.e., $\vec{c}$, and the rest of the set. $\hat{r}^{\geq}$'s guard can be computed by:

$$\forall j < k . \hat{r}_{gj}^{\geq} \equiv sel ? 0 : ((\vec{i} = j) \vee ((j > \vec{i}) \wedge c_j) \vee ((j < \vec{i}) \wedge c_{\vec{i}}))$$

To reduce the size of the domain for the value field, here we use $\langle 0, 1 \rangle$ to encode every $\mathbf{X}$. This way,

$$\hat{r}_v^{\geq} = sel ? \vec{c} : (1, \dots, 1)$$

In conclusion, we successfully reduced the value domain size from 2^k to at most $1 + 2 = 3$ (1 for the $<$ partition and 2 for the $\geq$ partition). The size of the value domain is at most 3 because there could be simplifications for some values of $\vec{c}$ (see the example below).

Example 5.2.5. $lt(\vec{n}, (1, 0, 1, 0)) = \vec{n} < (1, 0, 1, 0)$ partitions $\mathbb{B}^4$ into

1.

$$\begin{aligned} &\{(0, 0, 0, 0), (0, 0, 0, 1), (0, 0, 1, 0), \\ &\quad (0, 0, 1, 1), (0, 1, 0, 0), (0, 1, 0, 1), \\ &\quad (0, 1, 1, 0), (0, 1, 1, 1), (1, 0, 0, 0), \\ &\quad (1, 0, 0, 1), (1, 0, 1, 0)\} \end{aligned}$$

2.

$$\begin{aligned} &\{(1, 0, 1, 0), (1, 0, 1, 1), \\ &\quad (1, 1, 0, 0), (1, 1, 0, 1), \\ &\quad (1, 1, 1, 0), (1, 1, 1, 1)\} \end{aligned}$$

Based on Table 5.1 and Table 5.2, they are abstracted by the following ternary sets respectively:

1.

$$\{(0, \mathbf{X}, \mathbf{X}, \mathbf{X}), (0, 0, 0, 0), \\ (\mathbf{X}, 0, 0, 0), (0, 0, 0, 0)\}$$

2.

$$\{(1, 0, 1, 0), (1, \mathbf{X}, 1, \mathbf{X}), \\ (1, \mathbf{X}, \mathbf{X}, \mathbf{X}), (1, \mathbf{X}, 1, \mathbf{X}), \\ (1, \mathbf{X}, 1, 1)\}$$

The first ternary set can be simplified to $\{(0, \mathbf{X}, \mathbf{X}, \mathbf{X}), (\mathbf{X}, 0, 0, 0)\}$ because $(0, 0, 0, 0)$ is abstracted by $(0, \mathbf{X}, \mathbf{X}, \mathbf{X})$. The second ternary set can be simplified to $\{(1, \mathbf{X}, \mathbf{X}, \mathbf{X})\}$ because all other ternary vectors are abstracted by $(1, \mathbf{X}, \mathbf{X}, \mathbf{X})$. By using $\langle 0, 0 \rangle$ to encode $\mathbf{X}$ in the first ternary set and $\langle 0, 1 \rangle$ to encode $\mathbf{X}$ in the second set, the value field becomes $\{(0, 0, 0, 0)\}$ and $\{(1, 1, 1, 1)\}$.

Lemma 11. $\hat{r}^<$ and $\hat{r}^{\geq}$ together make up a data abstraction for $\mathbb{B}^k$ given the predicate $lt(\vec{n}, \vec{c}) = \vec{n} < \vec{c}$.

Proof. $\hat{r}^<$ and $\hat{r}^{\geq}$ satisfy the effectiveness requirement because the size of the union of their domain is exponentially smaller than the original domain. Let us show that it also satisfies the correctness requirement. Let us work on the first partition as an example.

For any value $\vec{a}$ from $q \in \mathbb{B}^k$ that is less than $\vec{c}$, there is at least 1 position (let it be $\vec{m}$) such that a_i is 0, c_i is 1 and at any location that is higher than $\vec{m}$, $\vec{a}$ is no larger than the corresponding bit in $\vec{c}$. Such value $\vec{a}$ is abstracted by the ternary vector generated by $\hat{r}^<$ when $\vec{i} = \vec{m}$. Besides, it is easy to verify that given any assignment to $\vec{i}$ ($env_{\vec{i}}$), the ternary vector generated by $\hat{r}^<$ is smaller than $\vec{c}$: $\hat{lt}(\hat{r}^<(env_{\vec{i}}), \vec{c}) = 1$. As a conclusion, the correctness requirement is also satisfied. $\square$

5.3 Summary

In this chapter, we first introduce our ternary abstraction based data abstraction with a realistic CAM verification. With the example, we illustrate the intuitions behind our data abstraction. Next we formalize our data abstraction: it uses guard-value encoding and symbolic indexing to abstract a set of Boolean values with a symbolic ternary vector that must satisfy both of the effectiveness and correctness requirement. Lastly, we show how to use our data abstraction in common scenarios.

Chapter 6

Guard-value Memory Abstraction

In this chapter, we present our memory abstraction based on the guard-value encoding. We first introduce it with the CAM example. Then we formalize this idea as a form of the value transition function simplification that has been presented in Section 3.4, and show that it is more general than existing memory abstraction approaches. Last but not least, we provide guidelines and templates to make it easier to apply our memory abstraction.

6.1 A Motivating Example

In Section 5.1, we show how to use data abstraction to reduce the input state space for **wtag**, **rtag** and **deltag** based on the fact that we only care about whether their values are equal to $\overrightarrow{vtag}$ instead of the exact value. In this section, we make a new observations that for each slot of the CAM, we only care about the data field if its associated tag's value is $\overrightarrow{vtag}$. We show how to utilize this new observation to simplify the verification problem even more.

Considering that a typical CAM may have hundreds and even thousands of memory slots, we will reduce the state space considerably if for all slots, their data's value fields are made the same. We can achieve this with the help of the value transition function simplification from Section 3.4.

The value transition function for the i -th slot's data (**data**[i]), i.e., $\text{Tr}_v(\text{data}[i])$, is shown in Listing 6.1. It is identical to the original transition function of **data**[i] except that it uses the corresponding value signals. For our CAM design, there are 3 cases for when the data field of the i -th slot is updated with **wdata**:

1. the slot's tag matches the write operation's tag (**wtag**);
2. the write operation's tag is absent in the CAM and the slot is the first available slot.

3. the write operation and delete operation happen at the same time. **wtag** is absent in the CAM and there is no available slot. The delete tag matches the slot's tag.

However, for simplicity, we neglect the details here and instead use **write_to_slot[i]** to denote the disjunction of the 3 cases.

Listing 6.1: The original value transition function for data in the i-th slot

```

1  if write_to_slot_v[i]                                // original Tr
2      data_v[i] = wdata_v                               // original Tr
3  else                                                  // original Tr
4      data_v[i] = data_v[i]                             // original Tr

```

According to the value transition function simplification, we need to construct $\hat{g}$ and Tr' for each slot i , denoted by $\hat{g}[i]$ and $\text{Tr}'[i]$, to abstract $\hat{\text{Tr}}(\text{data}[i])$ with

$$\hat{g}[i] ? \hat{\text{Tr}}(\text{data}[i]) : \text{Tr}'[i]$$

The corresponding value transition function for **data[i]** is:

$$g_v[i] ? \text{Tr}_v(\text{data}[i]) : \text{Tr}'[i]$$

Because we only care about the data of a CAM slot if its tag matches $\overrightarrow{vtag}$, we can use $\hat{\text{Tr}}(\text{tag}[i]) \hat{=} \overrightarrow{vtag}$ for $\hat{g}[i]$. As for Tr' , per our earlier analysis, we want to craft it in a way such that every CAM slot has the same data in the value verification. Provided that the 1) every slot's data is initialized to the same value and 2) there is at most 1 slot whose tag matches $\overrightarrow{vtag}$, we can achieve this by requiring that the data in each slot has the identical value transition function. Therefore, $\text{Tr}'[i]$ is

$$\bigvee_i (g_v[i] \wedge \text{write\_to\_slot}_v[i]) ? \text{wdata}_v : \text{data}_v[i]$$

$g_v[i] \wedge \text{write\_to\_slot}_v[i]$ is the condition for the slot whose tag is $\overrightarrow{vtag}$ to update its data with **wdata_v** in the new transition function for **data_v[i]**. The new guard transition function is $g_v[i] \wedge \text{Tr}_g(\text{data}[i])$. A better illustration of the new value transition function can be found in Listing 6.2. The new value transition function guarantees that each slot's data has identical value field. Its proof can be found in the next section, where we will also generalize this idea.

Listing 6.2: The new value transition function for data in the i-th slot

```

1  if Tr_v(tag[i]) =  $\overrightarrow{vtag}$                                 // g_v
2      if write_to_slot_v[i]                               // line 2-5: original Tr
3          data_v[i] = wdata_v                             //
4      else                                                //
5          data_v[i] = data_v[i]                           //
6  else

```

```

7   if  $\bigvee_i (\text{Tr}_v(\text{tag}[i]) = \overrightarrow{vtag} \wedge \text{write\_to\_slot}_v[i])$  // line 7-10: Tr'
8       datav[i] = wdatav //
9   else //
10      datav[i] = datav[i] //

```

In Table 8.3, it shows that memory abstraction helps the value verification so much that the value verification can be solved just by logic simplifications thus avoiding the costly SAT solving part of the verification. The memory abstraction becomes more effective for the CAM with larger data width parameter. For example, when the data width is set to 32, memory abstraction accelerates the verification 20 $\times$. As a result, using the memory abstraction, the verification time scales much better with the data width parameter of the CAM.

6.2 Formalization

Our memory abstraction is an application of the value transition function simplification. The main idea is to modify the data's value transition function such that every slot of the memory always has the same value. To use our memory abstraction, users need to have a modest understanding of the design and the property to verify to identify the memory that can be abstracted and provide $\hat{g}$ and Tr' for each memory slot. In Section 3.4.1, $\hat{g}$ for each memory slot is solely dependent on its location while in the previous section, $\hat{g}$ becomes more complicated as it depends on other signals' values. Our memory abstraction is more powerful than the existing approach such as [6] because it can abstract the memory such as CAM that cannot be done using the existing method.

From a correctness point of view, there is no requirement on $\hat{g}$ and Tr' because the next state of a slot computed using its new ternary transition function (3.6) is always an abstraction of the one using its original guard and value transition functions. However, in order to make our memory abstraction effective, $\hat{g}$ must be strong enough such that the memory slots satisfying their corresponding $\hat{g}$ are indispensable for the verification. The user-provided $\hat{g}$ for each slot could be dependent on any design signals, the slot's location, and Boolean variables from V . Our main contribution in this section is to provide a construction for a strong candidate for Tr' , which potentially saves users' effort in crafting their own.

Any array of elements is considered as a memory. An element is also called a slot. Memories are structurally repetitive by nature and their sheer sizes cause state spaces to explode and make verification problem unsolvable sometimes. Without loss of generality, we assume that the transition function for a slot is in the form of "if, else if, else if, ..., else". In the else-branch, the slot keeps its current value in the next cycle. A memory's repetitiveness implies that every slot's transition function has the same number of branches. For our convenience, we will represent a slot's transition function with 1) an array of conditions of the branches and 2) an array of expressions of signals that get assigned to the slots under the branches. For example, the i -th slot's transition function $\text{Tr}(\text{data}[i]) = c_0[i] ? e_0[i] :$

$c_1[i] ? e_1[i] : c_2[i] ? e_2[i] : \dots$ can be represented by $(c_0[i], c_1[i], \dots, c_{k-2}[i], 1)$ (the last branch's condition is 1 because it is the else branch) and $(e_0[i], \dots, e_{k-2}[i], \mathbf{data}[i])$.

Our candidates of Tr' are constructed in such a way that whenever a slot's value field satisfying its g_v changes its value in the next cycle, all slots not satisfying their respective g_v will be updated with that same value in the next cycle. Our candidates work best if at most one slot satisfies g_v in any cycle, in which case, using the provided Tr' will guarantee that every slot of the memory is equal to each other, effectively reducing the value field of the memory to just 1 slot. Tr' has the same structure as the memory's transition function, i.e., it also has k branches. We present two candidates for Tr' . They share the same array of branch conditions and differ in the array of assignments. The first (Section 6.2.1) is simple and has some strict restrictions on the memory it can handle but it is still powerful enough that it can be used for CAM. The other one (Section 6.2.2) relaxes the restrictions. Together, they are sufficient for the memory abstraction in all of our benchmark designs. They do not need a lot of gates to implement, therefore do not cause significant overhead in the problem size before logic optimizations.

6.2.1 A Candidate for Tr'

Let us consider a memory whose transition function satisfies the requirement that for any $l \leq k-2$, every slot's l -th assignment are the same, i.e., for any i and j , $e_l[i]$ is the same as $e_l[j]$. For example, in Listing 6.1, there are two branches. For the if-branch of every slot, it is assigned $\mathbf{wdata}$. For the else-branch of every slot, it keeps its old value. Therefore, the CAM satisfies our requirement. We now present our construction of $\text{Tr}'[i]$ for the i -th slot, represented by two arrays:

$$\left(\bigvee_j (g_v[j] \wedge c_{v_0}[j]), \dots, \bigvee_j (g_v[j] \wedge c_{v_{k-2}}[j]), 1 \right), \text{ and } (e_{v_0}[i], \dots, e_{v_{k-2}}[i], \mathbf{data}_v[i]) \quad (6.1)$$

$\text{Tr}'[i]$ defined in (6.1) has the following property.

Lemma 12. *If at most one slot's value field satisfies its corresponding g_v in any cycle, and the memory slots are initialized with the same value, then with the new value transition function based on Tr' specified in (6.1), the value field of every slot is equal to each other for every cycle.*

Proof. We prove it by induction. Let us assume that in the current cycle, every slot's value field is the same. We divide the proof into 2 cases. In the first case, no slot satisfies its g , therefore, i -th slot's value transition function $g_v[i] ? \text{Tr}_v(\mathbf{data}[i]) : \text{Tr}'[i]$ is reduced to $\text{Tr}'[i]$ and the else-branch of $\text{Tr}'[i]$ is taken. Therefore, the value field of i -th slot keeps its old value. and the lemma is also correct in the next cycle because of our inductive assumption that in the current cycle, every slot has the same value field.

In the other case, let us consider that 1 slot satisfies g_v and it is the i -th slot. Then, this slot's value transition function is reduced to its original one. Let us assume that the l -th branch is taken, therefore, the slot's value field in the next is $e_l[i]$. For any other slot, let it be j and $j \neq i$. Its value transition function is reduced to $\text{Tr}'[j]$. It is easy to see that in the transition function of the j -th slot, it is also the l -th branch that is taken. If the l -th branch is not the else-branch, slot j 's value field becomes $e_l[j]$ in the next cycle, which is equal to slot i 's value field $e_l[i]$ in the next cycle according to the requirement of using (eq:mem-abst-tr-basic) as Tr' . If the l -th branch is the else-branch, both of slot i and j will keep their current value in the next cycle. Based on the inductive assumption, it means that they will stay equal to each other in the next cycle. $\square$

6.2.2 A More General Candidate for Tr'

The requirements to use the Tr' candidate introduced in Section 6.2.1 could be too restrictive for some practical designs. For example, in the bypass design described in Section 8.1.6, the array of “number to add” (incr) has the transition function where there are branches under which the assignments (such as $\text{incr}[i] + \text{lkup\_val}[\text{tag}[i]]$) involve not only the slot itself ($\text{incr}[i]$) but also other signals that have dependency on the slot's location i , i.e., $\text{lkup\_val}[\text{tag}[i]]$. The signal lkup_val is a lookup table and $\text{lkup\_val}[\text{tag}[i]]$ is the number found at location $\text{tag}[i]$ in the lookup table. Because we care about the assignment only if it is part of the transition function of the slot that satisfies $\hat{g}$, we propose a candidate of Tr' that utilizes $\hat{g}[i](\hat{q}) = 1$ as a fact to simplify/transform the assignment in order to remove its dependency on i . For example, in the bypass example, if slot i satisfies $\hat{g}[i]$, then $\text{tag}[i]$ has Boolean values and is equal to $\overrightarrow{vtag}$. Thus, the assignment can be simplified to $\text{incr}[i] + \text{lkup\_val}[\overrightarrow{vtag}]$, which no longer depends on i .

Let us denote an operation $\downarrow (e_{v_j}[i], g_v[i])$ which simplifies $e_{v_j}[i]$ using the fact $g_v[i] = 1$ into a new value whose only dependency on i is via the slot i itself. If it can be done for every assignment in the transition function, then Tr' described in (6.2) can be used in the memory abstraction. Then, $\text{Tr}'[i]$ is

$$\begin{aligned} & \left(\bigvee_j (g_v[j] \wedge c_{v_0}[j]), \dots, \bigvee_j (g_v[j] \wedge c_{v_{k-2}}[j]), 1 \right), \text{ and} \\ & \left(\downarrow (e_{v_0}[i], g_v[i]), \dots, \downarrow (e_{v_{k-2}}[i], g_v[i]), \text{data}_v[i] \right) \end{aligned} \quad (6.2)$$

$\text{Tr}'[i]$ defined in (6.2) has the following property.

Lemma 13. *If at most one slot satisfies its corresponding g in any cycle, and the memory slots are initialized with the same value, then with the new value transition function based on Tr' specified in (6.2), the value field of every slot is equal to each other for every cycle.*

6.3 Summary

In this chapter, we introduce our memory abstraction and formalize it as a simplification of the memory's value transition function. The simplification for the memory write can be stated in the general form: the new value transition function of i -th slot becomes $g_v[i] \text{ ? } \text{Tr}_v(\text{data}[i]) \text{ : } \text{Tr}'[i]$. The key to successfully using our memory abstraction is to properly construct $\hat{g}[i]$ and $\text{Tr}'[i]$ for each slot i . We provide templates and guidelines for constructing $\text{Tr}'[i]$. Lastly, just like data abstraction, it is important to note that our memory abstraction does not always work well. Particularly, there are designs and properties whose correctness in each cycle involves a large number of memory slots. For such memory, using our technique could be detrimental because of the overhead caused by the ternary verification and the extra logic we would introduce with Tr' and g_v .

Chapter 7

Refinement

We present a framework for refinement in ternary verification. Since the monitor circuit for the property is combined with the design as a primary output, a ternary CEX is a ternary trace where the primary output is either 0 (strong ternary CEX) or X (weak ternary CEX). A strong CEX is spurious if it is because of illegal stimulus, i.e., the design is under-constrained. A weak CEX is spurious if the property passes for all Boolean stimuli that are abstracted by the weak CEX. A weak CEX is real if there exists a Boolean stimulus abstracted by the weak CEX that fails the property. Our framework is focused on weak CEXs. Spurious strong CEXs can be addressed by conventional Boolean refinement techniques. We will refer to spurious weak CEXs as spurious CEXs. The goal of the refinement is to eliminate spurious CEXs. Existing refinement approaches achieves this by increasing the precision of the ternary initial state and stimulus while we present a refinement framework that increases the precision of the unrolled implementation of the ternary transition function, which makes up the majority of a bounded model checking problem. In the next section, we will explain the differences in detail.

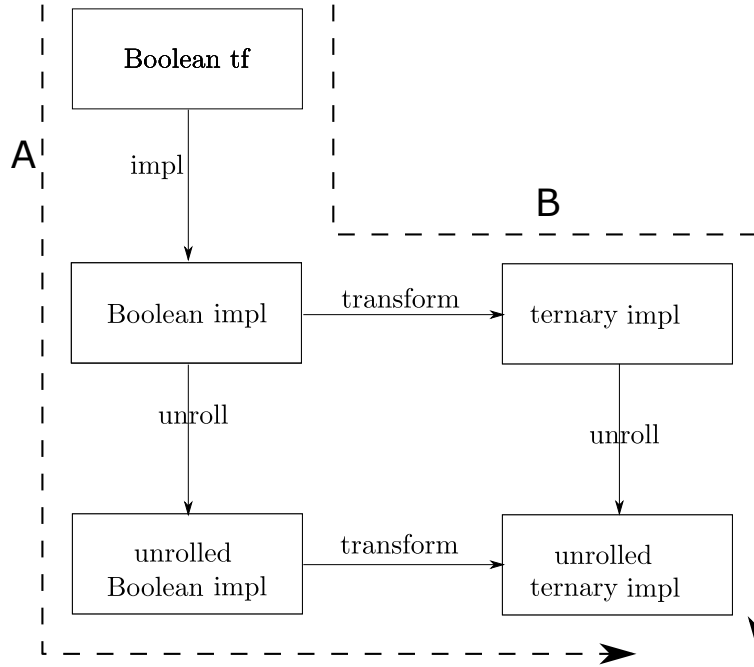
The most important part of the framework is to understand, from a novel point of view, why we encounter spurious CEXs in ternary verification: while existing approaches attribute the cause of the CEX be an insufficiently precise stimulus and/or initial state, we instead will show an alternative explanation, which is that the unrolled ternary implementation (short for “unrolled implementation of a ternary transition function”) for the primary output is not precise enough. Our refinement framework is built around this understanding of the spurious CEXs.

Recall that same as the existing work, we efficiently implement a ternary transition function by substituting each Boolean gate with its equivalent ternary gate (the “transform” step shown in Figure 7.1). The ternary verification problem can be viewed as checking whether the unrolled ternary implementation for the *primary output* is guaranteed to be 1. Instead of unrolling the ternary implementation, we equivalently unroll a Boolean implementation and replace the Boolean gates with the corresponding ternary ones. We prefer the construction of the unrolled ternary implementation this way (labeled with A in Figure 7.1)

because it helps us explain the differences between different refinement approaches within our framework later.

An unrolled ternary implementation is dependent on the unrolled Boolean implementation in the sense that for two unrolled ternary implementations defined using different unrolled Boolean implementations of the same Boolean transition function, they may return different values given the same ternary assignment to the inputs. In contrast, different unrolled Boolean implementations always return the same value given the same assignment to the inputs. In other words, for every unrolled Boolean implementation for the primary output, a corresponding unrolled ternary implementation is defined. The set of all unrolled ternary implementations is in fact a complete partially ordered set, ordered by the abstraction relation (Theorem 14). The bottom of the partially ordered set is the most precise implementation for which no spurious CEXs exist. Therefore, spurious CEXs are encountered because we choose a unrolled Boolean implementation whose corresponding unrolled ternary implementation is too abstracted. For example, Figure 2.1 and Figure 7.2 are different Boolean implementations of a multiplexer, but the ternary implementation defined using Figure 2.1 is an abstraction of the one using Figure 7.2 (compare the output using the stimulus $a = 1, b = 1, c = X$). The refinement approaches in our framework modify

Figure 7.1: Conventional (path ‘B’) and our way (path ‘A’) of constructing an unrolled ternary implementation from a Boolean transition function



the unrolled Boolean implementation. They are differentiated based on the scope of the modification and the type of the modification.

In the rest of the chapter, we will first differentiate our refinement from the existing ones in

Section 7.1. The basis of our refinement framework is presented in Section 7.2. Lastly, we will describe three different refinement methods within the framework in Section 7.3, 7.4 and 7.5. The first method straightforwardly follows Lemma 14. It modifies an unrolled Boolean implementation and is able to eliminate any spurious CEX. However, we believe it is challenging to use and prove the equivalence between before and after the modification due to the sheer size of the unrolled implementation. Our second method alleviates some of the challenges but focuses on only modifying a Boolean implementation (short for an implementation of the Boolean transition function), i.e., limiting ourselves to modifying the gates that are within the boundary of a design and the modification is repeatedly applied for every cycle. However, the efficiency comes at a cost of the completeness: there are spurious CEX that it cannot eliminate (discussed in Section 7.4.2). Our last method modifies the Boolean implementation in order to over-approximate as a means to partition the traces of the design. This method will introduce more Boolean variables in the verification but in return, it allows us to remove the spurious CEXs that the second method cannot.

7.1 Comparison with Existing Refinement Approaches

A spurious weak CEX is caused by a stimulus that assigns X to some input signals in some cycles and/or X to some registers in the first cycle (i.e., initial state). In their simplest form, existing refinement approaches (such as [44, 48]) use various ways to automatically or manually analyze a spurious weak CEX in order to find a small subset of such X assignments (will be denoted by signal-cycle pairs (n, c) in the thesis) that are responsible for the spurious CEX. In other words, if the X assignments in the subset are replaced with Boolean assignments, the spurious CEX would not occur. Some approaches are more careful and they also consider other signals' assignment and use it to qualify when the X assignments in the selected subset actually lead to spurious CEXs. They extend the signal-cycle pair to signal-cycle-Boolean condition triplet (n, c, p) . For example, consider a spurious CEX that assigns 1 and 0 to input signals a and b respectively in cycle 2, and X to the input signal d and f in cycle 3. After analyzing the spurious CEX, users figure out that assigning X to d in cycle 3 (i.e., the pair $(d, 3)$) causes the spurious CEX. Therefore, to fix it, the new stimulus will assign fresh Boolean variable to d in cycle 3. As for more careful approaches, let us consider that users further derive that only when a and b are assigned opposite values in cycle 2, assigning X to d in cycle 3 leads to spurious CEX, i.e., the triplet is $(d, 3, a^2 \neq b^2)$, if the original stimulus assigns the variable a^2 to a and b^2 to b in cycle 2. The new stimulus now will assign $(a^2 \neq b^2 \ ? \ v \ : \ X)$ to d in cycle 3, where v is a fresh Boolean variable. In general, a list of triplets could be needed for a refinement iteration, which will be used to assign symbolic values to different inputs or the same inputs but in different cycles or registers in the initial state.

Existing refinement approaches increase the precision of a ternary transition system by strengthening its ternary stimulus and initial state via replacing some X assignments with fresh Boolean variable assignments, which amounts to using symbolic variables to further

partitioning the input and initial state space. Let us consider the refinement triplets $(n_0, c_0, p_0), \dots, (n_{k-1}, c_{k-1}, p_{k-1})$, let the number of ternary stimulus satisfying the condition in every refinement triplet $(\bigwedge_{i < k} p_i)$ be m and the number of X assignments replaced be l . Then, the refinement partitions m ternary stimuli into $m \times 2^l$ ternary stimuli. Therefore, with each refinement iteration, the ternary verification is brought closer to the full-scale Boolean verification and the verification becomes more difficult. There are also some benefits for existing approaches. For example, they do not modify implementations of the transition function, and the stimulus and initial state are usually irrelevant to the exact implementation details of the transition function. It is useful in practice because hardware designs could be constantly evolving to fix bugs or implement new features. For the verification based on the guard-value encoding, existing approaches also make the guard verification simpler, however it is not enough to offset the significantly more time spent for the value verification (see Chapter 8.2).

Our refinement framework modifies the (unrolled) Boolean implementation instead of the ternary stimulus and initial state. It includes three methods that differ in the scope of the modification as well as the type of the modifications. We will use “re-implementation” to refer to modifying the implementation of a Boolean function without changing the functionality. The first method re-implements the unrolled Boolean transition function while the second re-implements the Boolean transition function. They will be presented in detail in the next few sections. Their most important benefit is that they do not partition the input stimuli nor the initial state thus retaining the same ternary input state space. But comparing with the existing approaches, they additionally need to check that the modifications are indeed re-implementations, which are not difficult especially for the second method because the modifications are incremental and local for each refinement iteration. The last method within the framework is more flexible by requiring that the modified Boolean implementation being an over-approximation instead of strictly re-implementation. The over-approximation partitions the traces by introducing primary inputs to some sub-circuits of the Boolean implementation, which are driven by Boolean variables in the ternary verification. The following example illustrates the third method.

Example 7.1.1. Consider a sub-circuit implementing $o = a \neq b ? a : b$, where, with the data abstraction, the domain for a and b is $\{X1, 1X, 00\}$. The ternary domain has enough information to decide whether the signals’ values are equal to 00. Without refinement, by assigning any value from the ternary domain that is not 00 to a and b , the select condition of the circuit ($a \neq b$) is X, which causes the sub-circuit’s output to become XX. To preserve the information needed for distinguishing from 00 at the output, our refinement method introduces a new primary input (let it be pi) and over-approximates the sub-circuit with $[(a \neq 00 \wedge b \neq 00) ? pi : a \neq b] ? a : b$. This way, whenever the values of a and b do not have enough information to decided whether they are equal, we replace $a \neq b$ with the primary input pi that is driven with a Boolean variables in the ternary verification, which essentially partitions the traces that assign XX to o :

- $(a = 1X, a = X1, o = XX)$ is partitioned into $\{(a = 1X, a = X1, o = 1X), (a = 1X, a = X1,$

$$o=X1)\}$$

- $(a=X1, a=X1, o=XX)$ is partitioned into $\{(a=X1, a=X1, o=X1)\}$
- $(a=1X, a=1X, o=XX)$ is partitioned into $\{(a=1X, a=1X, o=1X)\}$

Existing approaches partition traces of a ternary transition system by partitioning the input stimulus and initial state while our method partitions exactly at the parts of the traces where the information is lost. Existing approaches' partitioning affects much more logic than needed. Our refinement needs fewer Boolean variables thus has fewer partitions. In this thesis, we have not automated our refinement methods. However, as will become clear, provided the data abstraction, it is intuitive to locate the sub-circuits to modify.

7.2 A Complete Partially Ordered Set of Ternary Functions

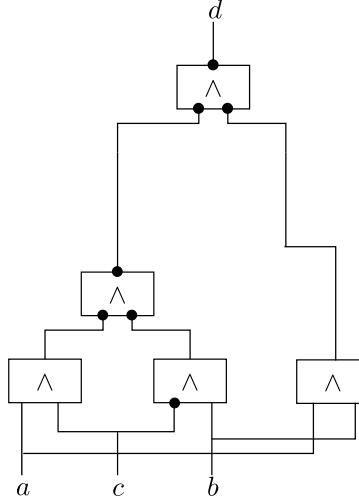
The unrolled Boolean (ternary) transition function for the primary output is a Boolean (ternary) function. For each signal, its Boolean (ternary) transition function is also a Boolean (ternary) function. Therefore, we will discuss in terms of ternary functions. A circuit implementation of a Boolean function is decided by the library of basic gates and the interconnections of the gates. When a ternary function, as an abstraction of a Boolean function, is defined by substituting Boolean gates in a circuit implementation of the Boolean function with the ternary ones, as shown in the following example, different implementations of the Boolean function will lead to different ternary functions. Lemma 14 shows that they can be partially ordered by the abstraction relation.

Example 7.2.1 (Different ternary MUX using different Boolean MUX implementations). *Figure 2.1 is the common implementation of a multiplexer. For this simple example, it is easy to find out the information is unnecessarily lost when the select line is X while the other inputs are both 1, in which case the ternary MUX based on Figure 2.1 outputs X instead 1.*

Figure 7.2 demonstrates an alternative Boolean implementation of a MUX whose corresponding ternary MUX outputs correct values when the select line is X and all other inputs are 1. It implements the formula $d = (a \wedge b) ? 1 : (c ? a : b)$, which is easy to prove the equivalence with $c ? a : b$.

The least abstract ternary function is given in Definition 18. The most abstract one is constructed in the proof of Lemma 14. By definition, for the least abstract ternary function, there is no spurious weak input assignment. An input assignment is spurious if the ternary function's output is X and after replacing X in the input assignment with fresh Boolean variables, the output of the ternary function always is the same Boolean value

Figure 7.2: A different AIG implementation of a multiplexer for abstraction refinement



regardless of the assignment to the fresh Boolean variables. We apply this result to the unrolled ternary transition function: the least abstract unrolled ternary transition function cannot have spurious CEXs.

Definition 18 (The least abstract ternary function). *The ternary function $\hat{f}^*$ abstracting the Boolean function f is the least abstract iff for any ternary assignment to its input $\hat{q}$,*

- $\hat{f}^*(\hat{q}) = 1$ iff for any Boolean input assignment that is abstracted by $\hat{q}$, the Boolean function returns 1, i.e., $\forall q \leq \hat{q} . f(q) = 1$.
- $\hat{f}^*(\hat{q}) = 0$ iff for any Boolean input assignment that is abstracted by $\hat{q}$, the Boolean function returns 0, i.e., $\forall q \leq \hat{q} . f(q) = 0$.
- $\hat{f}^*(\hat{q}) = \text{X}$ iff there exist two Boolean input assignments that are abstracted by $\hat{q}$, such that the Boolean function returns different values, i.e., $\exists q \leq \hat{q} . \exists q' \leq \hat{q} . f(q) \neq f(q')$.

Lemma 14. *Let $\mathbb{F} = \{f_0, f_1, \dots\}$ be the set of all possible circuit implementations of a Boolean function f , and let $\hat{f}_i$ be the ternary function that is defined using the implementation f_i . Then, the set of all ternary functions $\hat{\mathbb{F}} = \{\hat{f}_0, \hat{f}_1, \dots\}$ is a complete partially ordered set, ordered by the abstraction relation. $\hat{f}^*$ (Definition 18) is the least abstract one. The most abstract one returns X for any ternary input assignment where at least one input is assigned X.*

Proof. The proof is divided into 3 parts:

1. $\hat{f}^*$ is the least abstracted.
2. there exists an implementation of the Boolean function whose corresponding ternary function is the most abstracted.

3. there exists an implementation of the Boolean function whose corresponding ternary function is the least abstracted, i.e., equivalent to $\hat{f}^*$.

1. It is proved by contradiction. Assume that there is one ternary function $\hat{f}^?$ from $\hat{\mathbb{F}}$ that is less abstracted than $\hat{f}^*$, i.e., there is a ternary assignment where $\hat{f}^*$ returns $\mathbf{X}$ and $\hat{f}^?$ returns a concrete Boolean value. Recall that $\hat{f}^*$ only returns $\mathbf{X}$ for the ternary assignment where there exist two Boolean assignments that are abstracted by it and f returns different Boolean values for the Boolean assignments. Hence, $\hat{f}^?$ cannot be an abstraction of f , which contradicts with the assumption that that $\hat{f}^?$ is in the partially ordered set.

2. We construct an implementation of Boolean function, denoted by $f^\top$, that leads to the most abstracted ternary function. We construct $f^\top$ as a disjunction of all complete assignments to the inputs that make f output 1, i.e., a disjunction of minterms of f . $f^\top$ is obviously an implementation of f and for any ternary assignment where at least one input is assigned $\mathbf{X}$, its corresponding ternary function $\hat{f}^\top$ outputs $\mathbf{X}$.

3. We construct an implementation of Boolean function, denoted by $f^\perp$, that leads to the least abstracted ternary function. Let $\perp_i^+$ denote a min-DNF (Disjunctive Normal Form) for f . Since, a min-DNF is not unique, we differentiate them with the subscript, i.e., $\perp_0^+, \perp_1^+, \dots$. The implementation of the disjunction of all possible min-DNFs for f is denoted as $\perp_*^+$: $\perp_*^+ = \bigvee \perp_i^+$. Then, $f^\perp$ implements $\perp_*^+$. For any ternary assignment $\hat{q}$ such that $\hat{f}^*(\hat{q})$ outputs 1. There must exist a ternary term within a ternary min-DNF for f that evaluates to 1 for the ternary assignment. Therefore, $\hat{f}^\perp(\hat{q}) = \hat{\perp}_*^+(\hat{q}) = 1$. For any ternary assignment $\hat{q}$ such that $\hat{f}^*(\hat{q})$ outputs 0, there must exist an input whose assigned value makes a ternary term outputs 0 for any ternary term and any ternary min-DNF. Therefore, $\hat{f}^\perp(\hat{q}) = \hat{\perp}_*^+(\hat{q}) = 0$. In conclusion, $\perp_*^+$ is an implementation of f such that its corresponding ternary function is equivalent to $\hat{f}^*$. $\square$

7.3 Refinement by Re-implementing the Unrolled Boolean Transition Function

Illustrated in Figure 7.3, this refinement approach modifies the whole implementation of an unrolled Boolean transition function. It is a direct application of Lemma 14, which ensures that every spurious CEX can be removed eventually if each refinement iteration increases the precision of the corresponding unrolled ternary transition function. The following example will be used as running example to explain three refinement methods with regard to the completeness. A refinement method is considered complete if after finite refinement iterations, all spurious CEXs can be eliminated.

Example 7.3.1. *Let us consider a design with a vector of inputs ($\vec{a}$), 2 vectors of registers ($\vec{b}$ and $\vec{c}$) and a primary output (o). As convention, we use n' to denote the next-state*

value of the signal n . The definition of the transition functions for the system is:

$$\begin{aligned}\vec{b}' &= \vec{b} \neq \vec{a} ? \vec{a} : \vec{b}, \text{ // register} \\ \vec{c}' &= (\vec{b} = \vec{a} \wedge \vec{c} \neq \vec{a}) ? \vec{a} : \vec{c}, \text{ // register} \\ o &= \vec{b} \leq 5 \wedge \vec{c} \leq 5, \text{ // combinational}\end{aligned}$$

We verify the property that if $\vec{a}$ is no larger than 5 for 2 consecutive cycles, then the output must be 1 in the third cycle. Assume that we observe the following 3-cycle spurious CEX:

$$\begin{aligned}(\vec{a} = 00X, \vec{b} = 0XX, \vec{c} = 0XX, o = 1), \\ (\vec{a} = X00, \vec{b} = 0XX, \vec{c} = 0XX, o = 1), \\ (\vec{a} = 000, \vec{b} = XXX, \vec{c} = XXX, o = X)\end{aligned}$$

Based on the spurious CEX, $\vec{b} \neq \vec{a}$ and $\vec{b} = \vec{a}$ evaluate to X in the first cycle, $\vec{b}$ and $\vec{c}$'s values in the second cycle are both 0XX abstracting its if-branch and else-branch assigned values 00X and 0XX. In the second cycle, $\vec{b} \neq \vec{a}$ and $\vec{b} = \vec{a}$ also evaluate to X. Therefore, $\vec{b}$ and $\vec{c}$'s value in the third cycle is XXX abstracting its if-branch and else-branch assigned values X00 and 0XX. Neither of $\vec{b}$ and $\vec{c}$'s values in the third cycle has enough information to determine whether it is no larger than 5. Therefore, the output is X in the third cycle.

The refinement by modifying the unrolled Boolean transition function is challenging for this example. But on the other hand, it is a great example to demonstrate that if this refinement method is not done properly, checking the equivalence could be almost as difficult as verifying the property in the Boolean domain. Because the design is unrolled 3 times in this example, there 3 copies of $\vec{a}$. Let them be $\vec{a}^0$, $\vec{a}^1$ and $\vec{a}^2$. Similarly, the 3 copies of o are denoted by o^0 , o^1 and o^2 . The unrolled transition function for o is:

$$\begin{aligned}o^2 &= \vec{b}^2 \leq 5 \wedge \vec{c}^2 \leq 5, \\ \vec{b}^2 &= \vec{b}^1 \neq \vec{a}^1 ? \vec{a}^1 : \vec{b}^1, \\ \vec{c}^2 &= (\vec{b}^1 = \vec{a}^1 \wedge \vec{c}^1 \neq \vec{a}^1) ? \vec{a}^1 : \vec{c}^1, \\ \vec{b}^1 &= \vec{b}^0 \neq \vec{a}^0 ? \vec{a}^0 : \vec{b}^0, \\ \vec{c}^1 &= (\vec{b}^0 = \vec{a}^0 \wedge \vec{c}^0 \neq \vec{a}^0) ? \vec{a}^0 : \vec{c}^0\end{aligned}$$

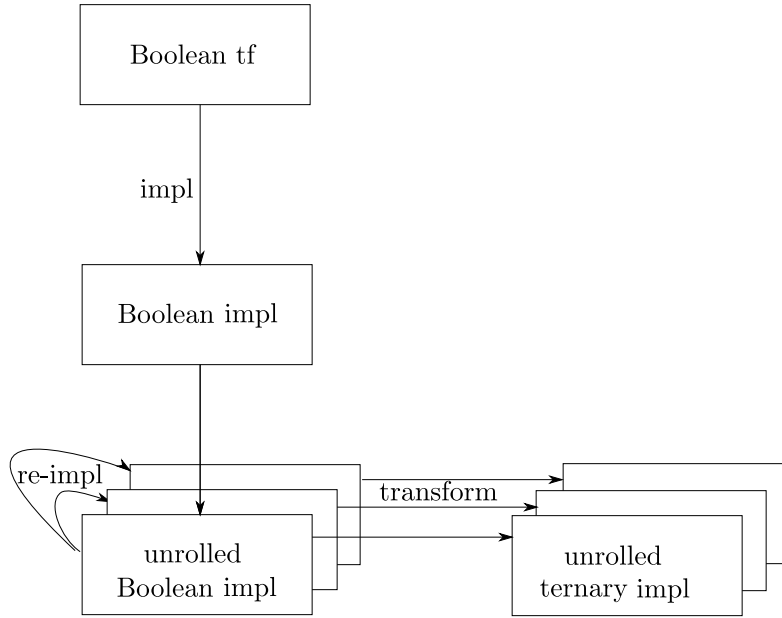
Based on the spurious CEX, we notice that in the first cycle both $\vec{b}$ and $\vec{c}$'s values are no larger than 5. For this small design, the input $\vec{a}$ is the only source where new data arrives, therefore it is straightforward to see that if initially $\vec{b}$ and $\vec{c}$ are no larger than 5 and $\vec{a}$ is no larger than 5 for 2 consecutive cycles, then in the third cycle, o is also no larger than 5. This way, we modify the unrolled Boolean transition function for o to (7.1) which can eliminate the observed spurious weak CEX.

$$o^2 = (\vec{b}^0 \leq 5 \wedge \vec{c}^0 \leq 5 \wedge \vec{a}^0 \leq 5 \wedge \vec{a}^1 \leq 5) \vee o^2 \quad (7.1)$$

Proving that (7.1) is equivalent to the original unrolled Boolean transition function is essentially proving the original property under the condition that initially $\vec{b}$ and $\vec{c}$ are no larger than 5. Furthermore, the new unrolled Boolean transition function implementation cannot eliminate all spurious CEXs, as we will encounter the ones where initially one of the $\vec{b}$ and $\vec{c}$ is larger than 5.

In Section 7.5, we will show how our third refinement method can tackle it effectively and properly.

Figure 7.3: Refinement by re-implementing the unrolled Boolean transition function



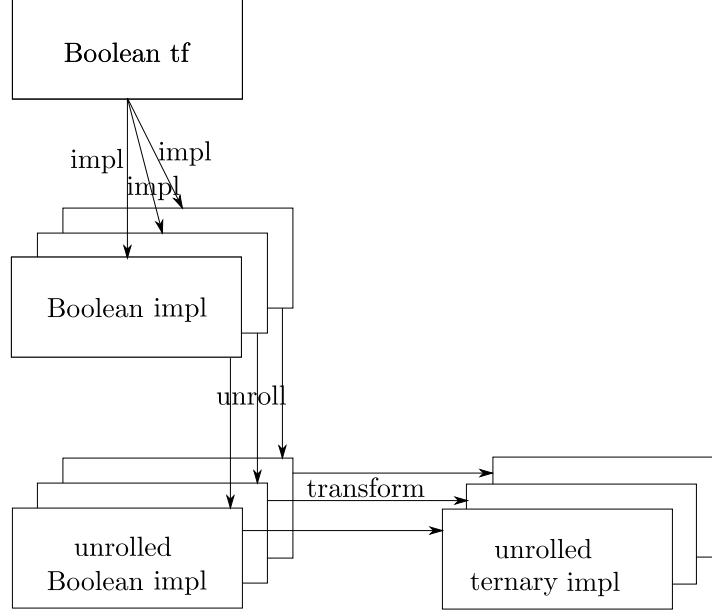
In practice, the unrolled transition function could be quite large especially as the bound increases, which we believe that it makes it more challenging to find the proper modifications that eliminate many spurious CEXs and remain easy to prove the equivalence (as demonstrated in Example 7.3.1). In the next section, we will restrict ourselves to only modify the implementation of the Boolean transition function. This way, it becomes easier to analyze the structure of the implementation that causes the spurious CEXs and it also becomes easier to prove the equivalence.

7.4 Refinement by Re-implementing the Boolean Transition Function

Illustrated in Figure 7.4, this refinement method re-implements the Boolean transition function. It improves the ease of use as well as alleviates the difficulty of the equivalence

checking of the re-implementation over the first method, but at the cost of completeness. In this section, we will first show how we use it to refine the data abstraction for the predicate $lt()$ from Section 5.2.4. Next, we will discuss about the completeness limitation: the spurious CEXs that this method intrinsically cannot eliminate.

Figure 7.4: Refinement by re-implementing the Boolean transition function



7.4.1 Refinement for the Less-than Data Abstraction

During our experiments, we found out that the implementation of a comparison between two numbers that use our data abstraction for $lt()$ is not precise enough. For example, the output of the implementation for $\mathbf{a} < \mathbf{b}$ outputs X instead 1 for $\mathbf{a} = \mathbf{X00}$. A typical implementation for $\mathbf{a} < \mathbf{b}$ outputs 1 if there exists a location i such that $\mathbf{a}_i = 0$, $\mathbf{b}_i = 1$ and for every index that is above i , $\mathbf{a}$ and $\mathbf{b}$ have the same value, i.e.,

$$\mathbf{a} < \mathbf{b} = \bigvee_i \left(\neg \mathbf{a}_i \wedge \mathbf{b}_i \wedge \left(\bigwedge_{j>i} \mathbf{a}_j = \mathbf{b}_j \right) \right)$$

By analyzing the implementation for $<$, it is clear that $\mathbf{X00} < \mathbf{101}$ outputs X because the highest bits of the two vectors, which are X and 1, do not have enough information to decide whether they are equal. We can modify the implementation for $\mathbf{a} < \mathbf{b}$ by realizing that it is only required that $\mathbf{a}_j$ and $\mathbf{b}_j$ cannot be 1 and 0 respectively. Therefore, our new

implementation of $a < b$ is:

$$a < b = \bigvee_i \left(\neg a_i \wedge b_i \wedge \left(\bigwedge_{j>i} \neg a_j \vee b_j \right) \right) \quad (7.2)$$

It is easy to verify that with the new implementation, $X00 < 101$ outputs 1 as expected.

7.4.2 Discussion of the Completeness

Per Lemma 14, refinement by re-implementing a Boolean transition function can only guarantee that there is no loss of information for a single signal's value and it cannot ensure the same for relations among multiple signals. In Example 7.3.1, there is no re-implementation such that $\vec{b}$'s value in the third cycle has enough information to compare it with 5 because its value XXX is already the most precise abstraction of the set $\{X00, 0XX\}$.

Therefore, when the X appearing at the output of the unrolled Boolean transition function can only be explained by the loss of information in the relation among multiple signals instead of them individually, re-implementing the Boolean transition function will not be able to resolve the spurious CEX. A feasible solution involves adding variables as described in the next section.

7.5 Refinement by Over-approximating the Boolean Transition Function

It is well known that the ternary abstraction loses information about the relation among signals when abstracting a set of states. For example, let us consider a set of Boolean states that map a and b to opposite values, i.e., $\{(1, 0), (0, 1)\}$. If we abstract this set with one ternary state, it maps a and b to (X, X) . When the relation among the signals is important to prove the property, we may need to further partition the ternary states, i.e., abstracting the set of Boolean states with more ternary states. In the above example, a proper partition to preserve the information is $\{(1, 0), (0, 1)\}$. As discussed in Section 7.1, the existing refinement approaches eliminate spurious CEXs by partitioning the input states and initial state for registers, which partitions the states of the other signals through simulation. In Section 7.4, we present how the refinement can be done more effectively by re-implementing the Boolean transition function when the spurious CEXs are caused by imprecision for individual signals' values. In this section, we consider the cases where ternary states partitioning is necessary and we show how to improve the existing approaches.

Instead of partitioning the input states and initial states, our approach partitions the states for the signals that explain the loss of information about the relations among the signals that are essential to prove the property. In Example 7.3.1, X at the output in the third cycle

is directly caused by the values of $\vec{b}$ and $\vec{c}$ not having enough information to determine whether they are no larger than 5, which can be explained by $\vec{b} \neq \vec{a}$ and $\vec{c} \neq \vec{a}$ being X in the previous cycle. Therefore, we partition the ternary state for $\vec{b} \neq \vec{a}$ and $\vec{c} = \vec{a}$ from $\{(X, X)\}$ to $\{(0, 0), (0, 1), (1, 0), (1, 1)\}$, which can be symbolically represented with two fresh Boolean variables. This way, in the third cycle, $\vec{b}$ and $\vec{c}$'s value can be either the if-branch value (0XX) or the else-branch value (X0X), both of which can be compared with 5.

The partitioning of the ternary states for internal signals is done by modifying their transition functions and creating new primary inputs in order to index the enlarged set of ternary states. In Example 7.3.1, $\vec{b} \neq \vec{a}$ only evaluates to X when both of $\vec{b}$ and $\vec{a}$ are no larger than 5 or both are larger than 5. But because the property to verify states that $\vec{a}$ is no larger than 5, we only need to consider the first case. Let p_0 and p_1 be 2 newly created primary inputs and they are driven with fresh Boolean variables in the stimulus. The modified $\vec{b} \neq \vec{a}$ is:

$$(\vec{b} \leq 5 \wedge \vec{a} \leq 5) ? p_0 : \vec{b} \neq \vec{a}$$

Similarly, the modified $\vec{c} = \vec{a}$ is:

$$(\vec{c} \leq 5 \wedge \vec{a} \leq 5) ? p_1 : \vec{c} = \vec{a}$$

In summary, in this refinement approach (illustrated in Figure 7.5), we first locate the signals ($\vec{n}$) whose state needs partitioning to eliminate spurious CEXs. This is done by analyzing the spurious CEXs backward from observed X. In our experiments, these signals are often part of the control logic. The partitioning is done by modifying their transition functions and creating new primary inputs, which are driven with fresh Boolean variables in the stimulus. Let the set of new primary inputs be $\mathbb{U}$. For our refinement method to be sound, the modified Boolean transition functions Tr' need to be an over-approximation:

$$\forall q . \exists q' \in \mathbb{U} \mapsto \mathbb{B} . (\text{Tr}(n_0)(q), \text{Tr}(n_1)(q), \dots) = (\text{Tr}'(n_0)(q, q'), \text{Tr}'(n_1)(q, q'), \dots)$$

Lastly, we will show where we used this refinement when verifying the benchmark designs.

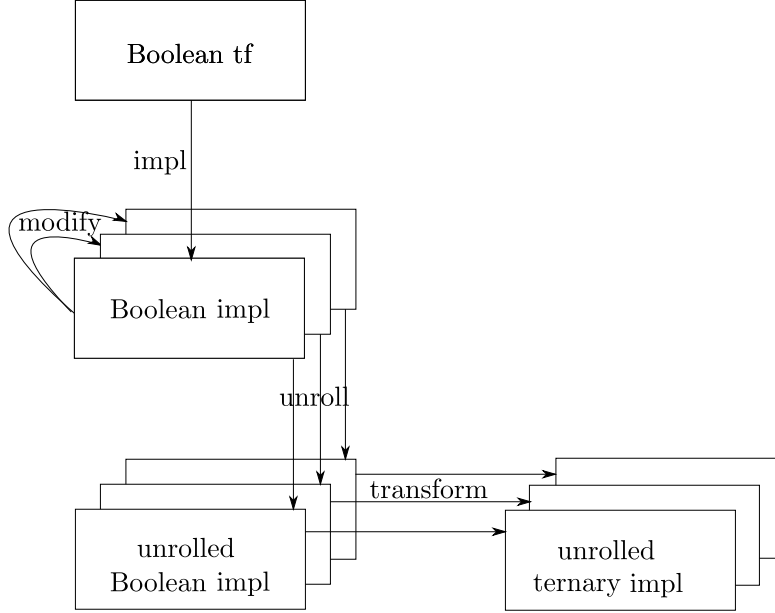
- If the data abstraction $eq(\vec{v})$ is applied to reduce the domain of $\vec{a}$ and $\vec{b}$, we partition the state space for $\vec{a} = \vec{b}$ when both are not equal to $\vec{v}$:

$$ref_{eq} : (\vec{a} \neq \vec{v} \wedge \vec{b} \neq \vec{v}) ? p_i : \vec{a} = \vec{b} , p_i \in \mathbb{U}$$

- If the data abstraction $lt(\vec{v})$ is applied to reduce the domain of $\vec{a}$ and $\vec{b}$, we partition the state space for $\vec{a} < \vec{b}$ when both are less than $\vec{v}$:

$$ref_{lt} : (\vec{a} < \vec{v} \wedge \vec{b} < \vec{v}) ? p_i : \vec{a} < \vec{b} , p_i \in \mathbb{U}$$

Figure 7.5: Refinement by over-approximating the Boolean transition function



7.6 Summary

We present 3 refinement methods that increases the precision of the ternary transition function by modifying Boolean functions instead of the ternary stimulus. This way, we show that they are able to retain more efficiency provided by the ternary abstraction than the existing approaches. The first two methods re-implement Boolean functions within the unrolled Boolean transition function and the Boolean transition function respectively. The last method over-approximates the Boolean functions within the Boolean transition function which effectively partitions the ternary states in the next cycle. We also explain how we use them in the verification of benchmark designs. Section 8.1.2 and 8.1.3 illustrate use of these techniques in case studies.

Chapter 8

Benchmarks and Results

In this chapter, we first describe 6 diverse designs and 8 properties that are used to benchmark our verification techniques. Then, we present the results and our analysis in details. Different techniques are compared in terms of the verification time and the size of the SAT problems generated. The results demonstrate the effectiveness of our techniques and back up our theories.

8.1 Benchmarks

We used the following circuits and properties to benchmark our verification techniques. The circuits cover a wide range of functionality. They imitate real-world circuits that are common for building a system yet challenging for existing formal verification techniques. The properties we verify are safety properties that describe the most important aspects of the circuits' respective functionality. For each property, its correctness requires almost every part of the circuit to work and interact with each other as intended. The properties are described in plain English for readability. In the implementation, each property is transformed into a monitor circuit that checks the signals of the design that are mentioned in the property. The monitor circuit's output is made a primary output of the design. A property is satisfied in the current cycle iff its monitor circuit's output is 1. Most of the designs in our benchmarks make extensive use of arrays that add considerable complexity to the formal verification. For each circuit and property, we provide the intuition behind how we were able to use data abstraction and/or memory abstraction.

The designs in our benchmarks are:

- CAM: a special type of memory where data is located by a tag instead of an address.
- Scheduler: an arbitration where the oldest one wins.

- Response buffer: using CAM to handle interleaved and out-of-order responses for each request.
- Shared resource manager: manage a collection resources identified by ID that can be shared by multiple consumers.
- Floating-point subtraction.
- Bypass: a synthetic example based on a superscalar microprocessor’s bypass paths.

Our data abstraction and memory abstraction are most applicable to circuits and properties where much of the complexity comes from the interaction of arrays and counters. Circuits without extensive use of arrays and counters might not benefit from our techniques. Fortunately, we have a method (Section 4.4) to inexpensively estimate the effectiveness of our techniques. We chose to create our own benchmark circuits, rather than using the verification problems from Hardware Model Checking Competition (HWMCC) [5], because the designs and properties in HWMCC are combined and represented as AIGs. This makes it impossible to understand the designs and properties, which is required to use our data abstraction and memory abstraction.

8.1.1 Content Addressable Memory

Refer to Section 5.1 for the description of our CAM design. Besides verifying the data correctness property of the CAM, we also verify its tag uniqueness property which states there are no two memory slots that are both valid and have the same tag. The verification of our CAM is more comprehensive and significantly more challenging than existing works that verify CAM using STE, e.g., [42, 44, 48], where the properties verified are inductive and the tag management logic is either irrelevant to the property or uses a simpler policy of inserting and deleting tags ([48]). Our verification covers more of the behaviour of the system and/or tackles more complicated CAM designs.

8.1.2 Scheduler

We implemented a scheduler based on Adams [2], where the oldest ready process gets to execute. Each process has an age register. The design is parameterized by age register width (OW) and the address width (AW). The number of processes is 2^{AW} . The property we verify states that the chosen process must be ready and older than all other ready processes. The property is inductive. We will use induction to fully verify it.

When verifying it, we arbitrarily pick a number by using symbolic variables $\vec{a}$ to be the largest age among the ready processes. For every ready process, we only care whether its age is smaller or equal to $\vec{a}$, which can be leveraged for data abstraction (see data abstraction for $lt()$ in Section 5.2.4).

As explained in Section 7.4.1, using the data abstraction for $lt()$ causes spurious CEXs. Our refinement modifies the implementation of the less-than comparison operation of the design to Formula 7.2. The existing refinement approaches, on the other hand, modify the data abstraction for $lt()$ to make it less abstract. We will provide more details in Section 8.2.4, where our refinement is compared with existing ones that are labelled with “alt data abst” in Table 8.7.

8.1.3 Response Buffer

The response buffer (RBuf) mimics practical designs where the responses to requests take unpredictable number of cycles and for each request, its response data is divided into a few fragments. RBuf temporarily stores the received data for each request until all fragments are received. Each request has a unique ID and it is recycled once all responses to the request has been received.

The response buffer is parameterized by `ID_WIDTH` which is the bit-width of requests’ IDs, `DATA_WIDTH` which is the total length in bits of a response data, `RESP_DATA_WIDTH` which is the bit-width for a response fragment and `TBL_SIZE` which is the number of `DATA_WIDTH`-bit wide memory slots the design has to buffer responses. The request ID ranges from 0 to $2^{\text{ID\_WIDTH}} - 1$. A request is in-flight if it is accepted by RBuf but has not been responded. `TBL_SIZE` decides the maximum number of in-flight requests.

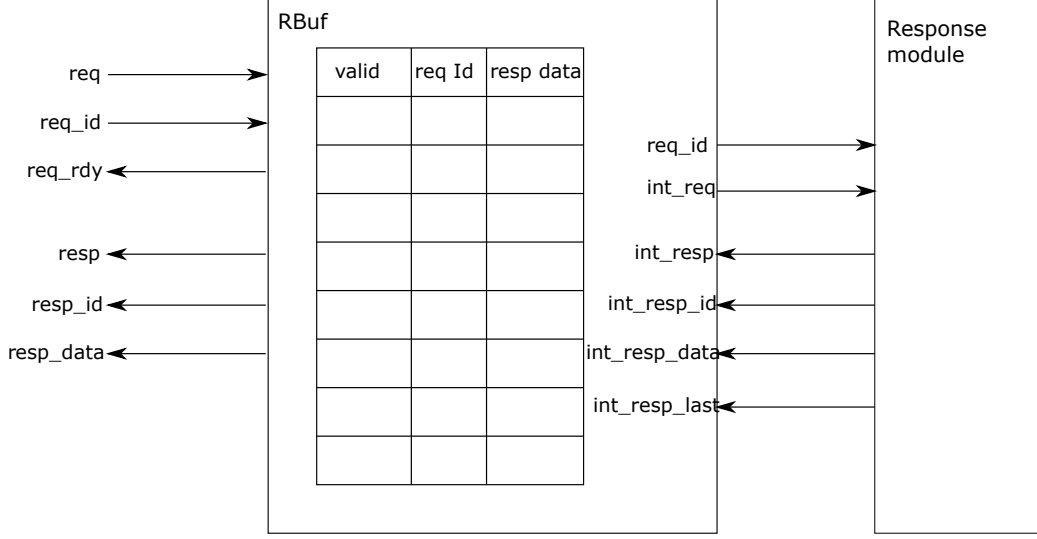
The schematic of RBuf is shown in Figure 8.1. The requestor sends a request id (`req_id`) to the RBuf. Response module divides the response into 4 fragments and sends them back at unpredictable time. A response fragment is valid if `int_resp` is 1, and it includes data (`int_resp_data`), its identifying ID (`int_resp_id`) and a flag (`int_resp_last`) indicating whether it is the last response fragment. There are other important interface and internal signals that we describe as follows. `req_rdy` indicates that RBuf has available memory to store the response data. Requests and responses are recorded in a table with 3 fields, which are request id, valid flag and response data. RBuf sends out responses together with the ID of the associated request (`resp_id`) when it has received all of the responses for the request, indicated by `int_last_resp` of the response module. The response module models the unpredictable delay to process and respond to each request. Besides, fragments of the responses to different requests can be interleaved. The two properties we verify are:

1. data correctness: for each request, the response data must be correct, i.e., it is a concatenation of the fragments sent by the response module for that request.
2. ID uniqueness: no in-flight requests can share the same ID.

Data Abstraction and Memory Abstraction

When verifying both properties, we arbitrarily pick an ID with symbolic variables $\vec{a}$. For the data correctness property, $\vec{a}$ is the ID of the request whose response data is to be

Figure 8.1: Response buffer



checked. For the ID uniqueness property, we verify that $\vec{a}$ is the ID of at most one in-flight request. For each request, we only care about whether its ID is equal to $\vec{a}$, which is an observation that can be leveraged for data abstraction to reduce the state space of the signals relevant to request/respond IDs. However, unlike the data abstraction in the CAM design, the signals here are not inputs. As a result, we need to manually modify the transition function for the signals such that its domain becomes $\{\vec{a}, \hat{\vec{a}}\}$, where $\hat{\vec{a}}$ denotes the symbolic ternary vector that generates every Boolean vector that is not equal to $\vec{a}$. The transformation may require some human expertise. In RBuf, the transition function for the request ID is described in the Listing 8.1 assuming ID_WIDTH=2. The transformed transition function is described in Listing 8.2. We construct the new transition function by syntactically constructing the condition for the request ID to be equal to $\vec{a}$. We leave out showing how to modify the transition function for the response ID because it is done similarly.

Listing 8.1: The original value transition function for req_id

```

1 req_id = 0
2 for (i=1; i<4; i++):
3     if req_id_avail[i] && !req_id_avail[i-1:0]
4         req_id = i

```

Listing 8.2: The new transition function for req_id

```

1 if (!verif_id && (!req_id_avail || req_id_avail[0])) ||
2     (verif_id && req_id_avail[verif_id] &&
3     !req_id_avail[verif_id-1:0])
4     req_id = verif_id # verif_id is assigned  $\vec{a}$  in the stimulus
5 else
6     req_id = ne_verif_id # ne_verif_id is assigned  $\hat{\vec{a}}$  in the stimulus

```

Along with the data abstraction, we can also use bit-width reduction for the value verification of both properties.

Furthermore, we do not care about the table entries whose request IDs are not equal to $\vec{a}$. As a result, we can apply memory abstraction for the valid flag field as well as the response data field of the table. But note that unlike the response data field, the valid flag field cannot be abstracted with $\mathbf{X}$ to prevent spurious CEXs. We also do not care about how the data is produced in the response module to respond to the requests whose IDs are not equal to $\vec{a}$, which allows us to apply memory abstraction to the relevant array in the response module.

Refinement

We need to apply the refinement for “equal to a vector” ref_{eq} introduced in Section 7.5 when comparing `req_id`, `resp_id` with the ID number ranging from 0 to $2^{\text{ID\_WIDTH}} - 1$ and comparing the request ID in the table of RBuf with `resp_id`. `req_id` and `resp_id` are the signals where we directly apply the data abstraction while the request IDs in the table are data-abstracted implicitly because their values are decided by logics that use `req_id` and `resp_id` and their domain is also the abstracted data domain. If we use the existing refinement approaches that are restricted to modifying the antecedent, `req_id` and `resp_id` must always have Boolean values because, otherwise, the signals that track whether the request IDs that are not equal to $\vec{a}$ are available or not have the value $\mathbf{X}$, which leads to the smallest available request ID being all $\mathbf{X}$ s. With the existing refinement approaches, the ternary verification is reduced to Boolean verification. As a result, the guard verification becomes trivial, as shown in Table 8.14.

8.1.4 Shared Resource Manager

Resources such as memory can be shared by multiple consumers. The basic operations of the shared resource manager include handling the requests for resource allocation from the consumers and reclaiming the resources returned. The manager should also indicate to the consumers whether it has any resource left to allocate. The property we verify states that each resource cannot be given to more than 1 consumer. The policy for allocation and reclaiming is highly customizable and can be very complex. We implement a simple manager where we assume that only one consumer can request or return resources in each cycle and each consumer returns the resource that is least recently requested by that consumer.

Data Abstraction

When verifying the property, we arbitrarily pick a resource identified by $\vec{a}$, then we use data abstraction based on the predicate $eq(\vec{n}, \vec{a}) \equiv \vec{n} = \vec{a}$ to abstract away the information

only needed to distinguish among resources that are not $\vec{a}$. For the design, available resources are stored in a FIFO. Initially, every resource $(0, 1, \dots, k-1)$ is available. Thus, the initial states of the memory in the FIFO before and after data abstraction are shown in Figure 8.2. Recall that $\neq \vec{a}$ is a symbolic ternary vector that compactly abstracts the set of Boolean vectors that are not equal to $\vec{a}$ as shown in Formula 5.3.

Figure 8.2: Initial state of the FIFO storing available resources

Original			
0	1	...	$k-1$

With data abstraction			
$(\vec{a} = 0) ? \vec{a} : \neq \vec{a}$	$(\vec{a} = 1) ? \vec{a} : \neq \vec{a}$	...	$(\vec{a} = k-1) ? \vec{a} : \neq \vec{a}$

8.1.5 Floating Point Subtraction

A floating point number can be represented by $1.\text{mantissa} \times 2^{\text{exponent}}$, where mantissa and exponent are Boolean vectors. In our design, the inputs are aligned, i.e., they have the same exponent, which is usually the first step in floating point subtraction algorithm. The result of $1.m_1 \times 2^e - 1.m_2 \times 2^e$ is computed by shifting the result of $(m_1 - m_2)$ to the left until there is no more leading zeros and the exponent of the result is decreased accordingly. For example, $1101 \times 2^{11} - 1010 \times 2^{11} = 0011 \times 2^{11} = 1100 \times 2^{01}$. Our design is optimized for performance by using pipeline to implement the left-shift of $m_1 - m_2$. Our property checks that the mantissa of the subtraction is correct. The property is inductive and we fully verify the property using induction.

For the convenience of applying the data abstraction, we decompose verifying the design into verifying the gates computing $m_1 - m_2$ and verifying the gates shifting $m_1 - m_2$ to the left until the leading 0s are gone, which is the main complexity of the design. The control logic of our design only cares about the number of leading 0s. Therefore, we make 2 copies of $(m_1 - m_2)$. One copy is used in the data path while the other copy is used for control logic. For the copy used for the control logic, we can abstract it by $\{(1, X, \dots, X), (0, 1, X, \dots, X), \dots, (0, \dots, 0, 1)\}$, which can be represented by $\langle a_{g_j}, a_{v_j} \rangle : a_{g_j} \equiv \vec{v} \geq j$ and $a_{v_j} \equiv \vec{v} = j$ using the guard-value encoding, where $\vec{v}$ is a vector of Boolean variables used as the indexing variables from 0 to $k-1$, and k is the length of $m_1 - m_2$. The ternary vectors in the set abstracts the Boolean vectors with 0, 1, and $k-1$ leading 0s. Our two copies of $(m_1 - m_2)$ and the additional gates used to process them in

the pipeline explains why in Table 8.22, Roorda has a lot more clauses than the BMC. No refinement is needed.

8.1.6 Bypass

Bypassing is an optimization technique that is commonly used in microprocessors to reduce the latency of executing instructions. With bypassing, one instruction can directly get the latest values of its operands from the previous instruction that modifies the operands but is still in the pipeline, instead of waiting for the previous instruction to exit the pipeline and write the data to the memory so that latest value can be read from the memory for the instruction to be executed. Bypassing is one of the micro-architectural features that is particularly challenging to verify.

Our bypass design (shown in Figure 8.3) implements the same idea but with some simplifications and modifications. The purpose of the modifications is to increase the verification complexity. The “instructions” to our design only has one field which is the address where it updates the data in the memory and the only operation it has is to add the data at the address of the memory with the symbolic value that is looked up based on the last 3 bits of the instruction address. The design has 5 stages. Depending on the last 3 bits of the instruction address, the addition could happen in one of the stages. Furthermore, bypassing can happen even before the addition is executed. As a result, there is a table that keeps record of how much to add for each instruction when it reaches the execution stage. The memory is dual-port supporting concurrent read and write. If reading from the same address as the write, the write data is to be read.

Figure 8.4 shows two snapshots of the pipeline and the table’s states in cycle 0 and cycle 3. Two instructions with the same address 0XF3 arrive 3 cycles apart. The instructions’ execution stage is 4th stage and the number to add is 0X02. When the second instruction enters the pipeline at cycle 0, the earlier instruction is yet to exit its execution stage, where the operands are 0XB0 and 0X02. 0XB0 is the data read from the memory at the location 0XF3 when the earlier instruction enters the pipeline. Because the addition is yet to happen, 0XB0 is bypassed to the first stage and accordingly, in the table, the entry for the instruction is updated to $0X02 + 0X02 = 0X04$. In cycle 3, the operands to the execution stage becomes 0XB0 and 0X04. When the second instruction exits the pipeline, it will write 0XB4 to the memory.

8.2 Results

In this section, we demonstrate the effectiveness of the data abstraction, the SAT-based implementation of STE using guard-value encoding and logic optimization, the guard-value memory abstraction and our refinement approach by presenting the results when verifying properties of the benchmarks introduced in the previous section. We will first discuss about

Figure 8.3: Bypass design

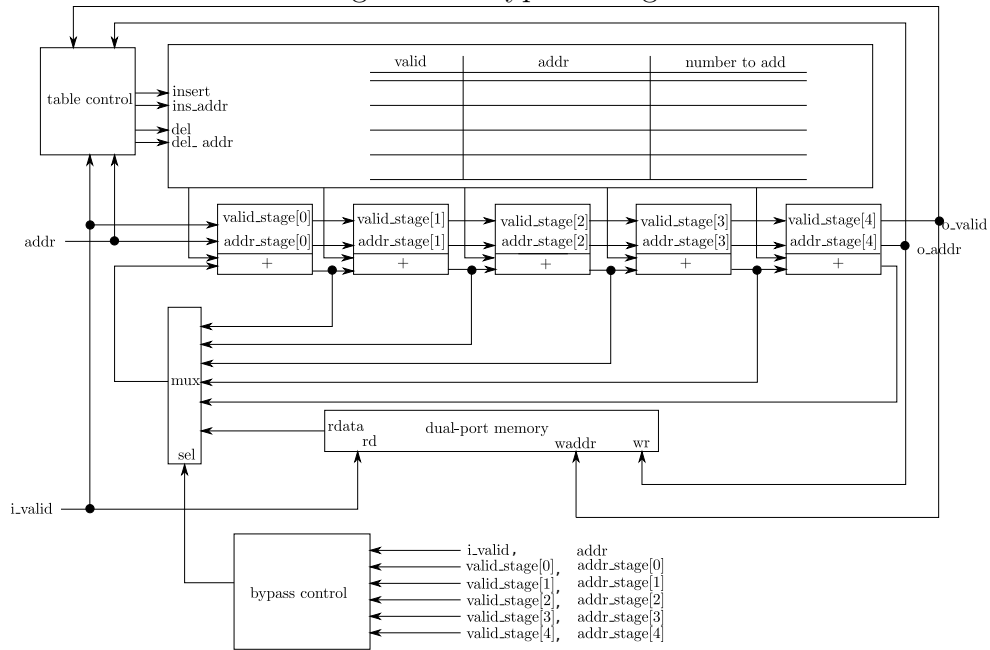
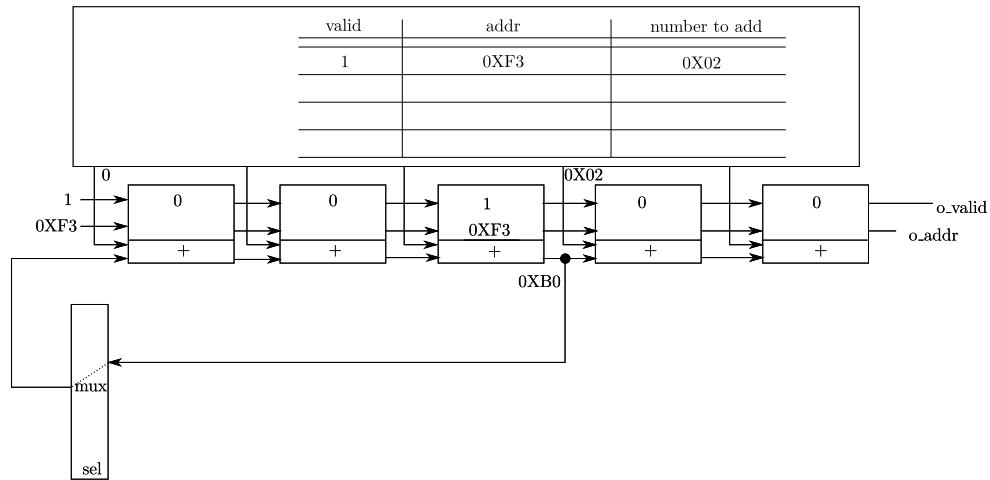
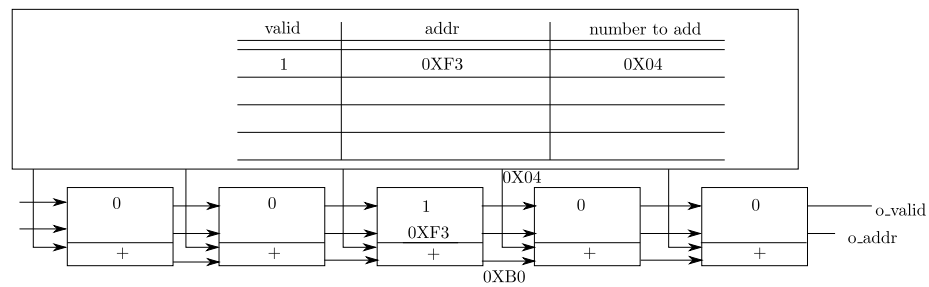


Figure 8.4: A run of the bypass

Cycle 0



Cycle 3



the design of the experiments. Then, an overview of the results is provided. Lastly, details results are given for each benchmark design.

8.2.1 Experiments Setup

In this thesis, we focus on verifying safety properties. If they are not k -inductive, we will verify that they are correct up to a certain bound. As mentioned in the previous section, a safety property is compiled into a circuit whose primary output is $\mathbf{o}$. This way, any safety property is transformed to that $\mathbf{o}$ is always 1. Because our definition of a ternary transition system $(\hat{\mathcal{M}})$ includes the symbolic ternary initial state $(\hat{\mathcal{Q}}_R)$ and symbolic ternary stimulus $(\hat{\sigma}_I)$, there is a straightforward way to construct an antecedent to convert a safety property $\mathbf{G}(\mathbf{o} = 1)$ to an STE property while preserving the bounded semantics (let the bound be k):

$$\bigwedge_{n \in S_R} n \text{ is } \hat{\mathcal{Q}}_R(n) \wedge \bigwedge_{n \in S_I} n \text{ is } \hat{\sigma}_I^0(n) \wedge \dots \bigwedge_{n \in S_I} \mathbf{N}^{k-1} n \text{ is } \hat{\sigma}_I^{k-1}(n) \implies \mathbf{N}^{k-1} (\mathbf{o} \text{ is } 1) \quad (8.1)$$

Checking this STE property (8.1) is equivalent to verifying the safety property $(\mathbf{G}(\mathbf{o} = 1))$ using a ternary BMC. Both algorithms use ternary symbolic simulation to compute the reachable states for $\mathbf{o}$ in cycle k based on the same initial state and stimulus. In other words, in the settings of this thesis, for any particular ternary encoding, ternary BMC is equivalent to STE. For example, BMC-GV is equivalent to STE-GV. As a result, by comparing the effectiveness of ternary BMC based on different encodings (BMC-GV, BMC-DR and BMC-Roorda), we are also comparing STE-GV, STE-DR and STE-Roorda. Because of the dual purposes of the experiments of BMC-GV, BMC-DR, and BMC-Roorda, we label them with GV, DR and Roorda respectively in this section.

Recall that data abstraction modifies the stimulus and initial states of $\hat{\mathcal{M}}$ for some signals. When we compare GV, DR and Roorda, they always verify the same $\hat{\mathcal{M}}$. If memory abstraction that can be implemented in the Boolean domain is used (e.g. in the Bypass verification), we will implement it in both of $\mathcal{M}$ and $\hat{\mathcal{M}}$ the same way in order to be fair to BMC. Logic optimizations are used right before translating AIGs to CNF and calling miniSAT [24] for GV and DR most of the times. Exceptions will be noted. Our logic optimizations are composed of 3 ABC functions that are the cores of their respective ABC command wrappers `drwsat`, `dfraig` and `&dc2`. The parameters of the logic optimization algorithms are static and no effort was put into adjusting the parameters to achieve best performance. We use the state of art implementation of BMC (command `bmc3`) in ABC. Because BMC and Roorda do not use symbolic simulation and their verification problem is never represented as an AIG, logic optimizations are not applicable.

For every design, we use GV, DR, Roorda and BMC to verify the properties. GV, DR and Roorda all implements the same data abstraction, but extra care is taken into optimizing the value field as part of the data abstraction for GV. The properties we verify for each

design are high-level properties that specify the most important aspect of their functional correctness. For the properties to be satisfied, all major logics of the design need to behave correctly. These properties are challenging for existing verification techniques as will become clear. We insert bugs that are difficult to find into the shared resource manager and the bypass in order to test GV’s ability of catching bugs. Particularly, the bug in SRM can only be detected by GV and all other techniques (DR, Roorda and BMC) timed out. For CAM, bypass and RBuf, GV memory abstraction is applicable. We will compare the performance of GV with and without using GV memory abstraction. The bit-width reduction enabled by the guard-value encoding is applicable for CAM, RBuf, SRM and bypass. Scheduler, RBuf and bypass are the designs where we tried existing ternary refinement approaches and compared with our approach.

We use the following abbreviations when reporting results in the tables:

- cl: clause.
- v cl#, g cl#: the number of clauses in the value verification and guard verification respectively.
- vAND, gAND: and in the value verification and in the guard verification respectively.
- oTime, sTime, vtime and time represent optimization time, SAT solving time, value verification time and the total time respectively.

For BMC and the ternary verification using Roorda’s implementation, only the number of clauses is available. Therefore, when comparing against these two techniques, we will compare the problem’s size in terms of the number of clauses instead of the ANDs in the AIG. There is a tight correlation between the number of ANDs and the number of clauses they translate to. Bit-width reduction is only possible with the guard-value encoding and it only simplifies the value verification while leaving the guard verification unaffected. As a result, we only report the value verification time when we study the effect of the bit-width reduction. When studying the gv memory abstraction, we observe that the memory abstraction usually causes slight increase in the problem size before the logic optimization. However, the memory abstraction provides more optimization opportunities, which enables the logic optimization to reduce the verification problem much more than without the memory abstraction. To show this phenomenon, we provide the number of ANDs for both before and after logic optimization.

We use Yosys [50] to compile Verilog. For BMC, safety properties are written as SystemVerilog assertions and we use SymbiYosys [49] to compile the design and assertions, and call `bmc3` to check the assertions. We set time-out limit to be 1800 seconds for each verification. The experiments were done using AMD Ryzen 7 3700X and 16 GB RAM.

8.2.2 Overview of Results

Figures 8.5 - 8.9 aggregate the experimental results in terms of the verification time and plot them against verification problems, which are determined by the design, parameters of the design, the property to verify and the bound (as well as the verification technique for Figure 8.9). The verification problems are mapped to numbers which are used to label x-axis in the figures. The details of this mapping can be found in Table 8.1. Note that Figure 8.7 and Figure 8.8 are meant to show the effectiveness of the techniques that are unique to GV, therefore the performance of DR, Roorda and BMC are not included. We make the following observations.

1. From Figure 8.5 and 8.6, GV significantly outperforms BMC in orders of magnitudes when verifying designs with or without bugs, because of the data abstraction and the guard-value encoding. Details of the outliers can be found in Table 8.10 and Table 8.7. In Table 8.10, it is shown that as soon as the bound increases, GV start to dramatically outperform again. In Table 8.7, GV performs worse than others only for the smallest bound where the verification problems are simple enough to be solved in seconds. GV is almost always faster than DR also in a few orders of magnitudes sometimes, because of the guard-value encoding and the extra don't-care optimizations for the value field. DR is dramatically better than Roorda likely because of the logic optimizations. Roorda lags behind BMC likely because of the considerable overhead in the extra clauses representing the ternary stimulus, where data abstraction is specified (see Table 8.2, 8.5, 8.6, 8.10, 8.15, 8.17, 8.20, 8.18, 8.22, 8.23 and 8.26 to compare the problem size for Roorda and BMC). More importantly, for many verification problems, GV is the only technique that can solve them without time-out.
2. From Figure 8.7, we observe that the GV memory abstraction is very effective. The speed up can be more than $8\times$. Besides, there is a verification problem that GV can only solve with the GV memory abstraction.
3. From Figure 8.8, we observe that the bit-width reduction consistently improves GV upto $4\times$.
4. From Figure 8.9, we can see that our refinement is better than the existing refinement approaches most of the time. Majority of the outliers are for DR and Roorda, which is not of main concern. The 2 outliers for GV will be explained in Section 8.2.4 and Section 8.2.5.

There are 5 types of tables, which provide more details to the results of our experiments and will be presented from Section 8.2.3 to 8.2.8.

1. The tables that compare GV, DR, Roorda and BMC. In these tables, the number of clauses and the verification time are reported. For GV, we only list the number of clauses of the *value verification* because it is the most relevant and important when

comparing with other techniques especially BMC. The number of clauses for GV and DR are acquired after the logic optimizations.

2. The tables that show the effect of using the GV memory abstraction. The sizes of the AIGs for both of the guard verification and value verification are compared between using and not using GV memory abstraction. The total verification time is also compared.
3. The tables that show the effect of using the bit-width reduction. The size of the AIG before logic optimization for value verification is compared between using and not using the bit-width reduction. The value verification time is also compared. Because the bit-width reduction does not modify the guard verification in any way, statistics relevant to the guard verification are not reported.
4. The tables that compare our refinement approach with the existing ones. The sizes of the verification problem before and after logic optimizations are provided for using our refinement approach and existing refinement approaches. The size of the verification problem for different techniques are measured differently. For GV, it is $\langle \# \text{ of g cl}, \# \text{ of v AND} \rangle$. It is $\# \text{ of AND}$ for DR, and $\# \text{ of cl}$ for Roorda. The verification time is also provided.
5. The tables whose purpose are not listed above. For example, there are tables which have results that we use to discuss the performance of adapting data abstraction to BMC.

These tables can be used with the figures to locate the outliers as we have shown. More importantly, we can use them to make further observations and conclusions.

1. Table 8.10, 8.15, 8.17, 8.18 and 8.23 show that GV scales significantly better than the other techniques (DR, Roorda, BMC) in terms of the clock-cycle bound. Table 8.5, 8.6 and comparing Table 8.17 with Table 8.18 unveil that GV scales dramatically better in terms of the design parameters as well. Recall that based on our analysis, the size of the value verification is smaller than the rest of the techniques. The results shown in the above tables and Table 8.2 align with our analysis and furthermore, we can see that logic optimizations can make the value verification become even smaller and sometimes solve it directly.
2. Table 8.3, 8.13 and 8.25 show that GV memory abstraction can further improve GV's scaling with the bound and the parameters. In Table 8.3, the verification time scales approximately linearly with the data width parameter. In these tables, we also provide the size of the guard verification before and after logic optimizations, after examining which, we can see that the size of the guard verification before logic optimization is much larger than the corresponding value verification but gets greatly simplified by logic optimizations and even solved in many cases. This supports our

analysis in Section 3.2 that the guard verification exhibits significant redundancy in logics and is a much simpler verification problem than the value verification. Besides, looking at the size of guard verifications in the tables, we can see that even though GV memory abstraction increases the size of verification before logic optimizations because of extra logic used to implement it in the design, GV memory abstraction brings additional logic optimization opportunities that can be utilized by off-the-shelve logic optimizations such that they become much smaller than without GV memory abstraction after logic optimizations.

3. Table 8.4, 8.16, 8.19 and 8.24 show that the bit-width reduction effectively reduces value verification’s size after logic optimizations, which indicates the bit-width reduction we use is sophisticated enough that the logic optimization algorithms are incapable to achieve. The improvement in size of the value verification and verification time is most significant for SRM design.
4. Table 8.9, 8.14 and 8.27 show that our refinement method works better than existing ones not only for GV but also for DR and Roorda most of the time. Existing approaches modify the stimulus to remove X , and sometimes remove X completely (e.g., RBuf). Therefore it is expected that the guard verification can be simpler with existing refinement approaches: it can be observed in the results that it takes longer for the logic optimizations and SAT solvers to simplify/solve the guard verification. This disadvantage of our refinement approach usually is more than overcome by the much simpler value verification of our refinement, maybe except for simple verifications. Based on our experiments, in almost all cases, the guard verification is simpler than the value verification and the guard verification scales better than the value verification in terms of bounds and design parameters. From the tables, we can also observe that using existing refinement, the size of the value verification as well as the size of the verification problem for DR and Roorda can become smaller due to the simpler stimulus that uses fewer gates. But in the meantime, it also operates at a less abstracted level than using our refinement, which turned out to be a more important factor than sizes when it comes to determining the verification time. From the results, we can see that both logic optimizations and SAT solvers can utilize the higher abstraction levels brought by our refinement to accelerate the verification, especially for GV and DR.

Table 8.1: The mapping from x-axis' labels to verification problems

#	circuit	parameter	property	bound
0	CAM	DW=2, TWIDTH=8	data correctness	18
1	CAM	DW=16, TWIDTH=8	data correctness	18
2	CAM	DW=32, TWIDTH=8	data correctness	18
3	CAM	DW=2, TWIDTH=8	tag uniqueness	2
4	CAM	DW=2, TWIDTH=16	tag uniqueness	2
5	CAM	DW=2, TWIDTH=32	tag uniqueness	2
6	Scheduler	AGE_WIDTH=8		2
7	Scheduler	AGE_WIDTH=16		2
8	Scheduler	AGE_WIDTH=32		2
9	RBuf		data correctness	6
10	RBuf		data correctness	8
11	RBuf		data correctness	10
12	RBuf		data correctness	12
13	RBuf		tag uniqueness	6
14	RBuf		tag uniqueness	8
15	RBuf		tag uniqueness	10
16	RBuf		tag uniqueness	12
17	SRM	consumer_number=2		6
18	SRM	consumer_number=2		8
19	SRM	consumer_number=2		10
20	SRM	consumer_number=2		18
21	SRM	consumer_number=4		6
22	SRM	consumer_number=4		8
23	SRM	consumer_number=4		10
24	FPSub			
25	Bypass			11
26	Bypass			15

A number in the format of $n.m$ can be found in Figure 8.9. The verification problem it refers to is n . The interpretation of m is:

- .0: guard-value encoding.
- .1: guard-value encoding and encX 0, which is explained in Section 8.2.4.
- .2: guard-value encoding and encX symb, which is explained in Section 8.2.4.
- .3: dual-rail encoding.
- .4: Roorda's encoding of ternary circuits.

Figure 8.5: GV is faster than DR, Roorda and BMC at proof

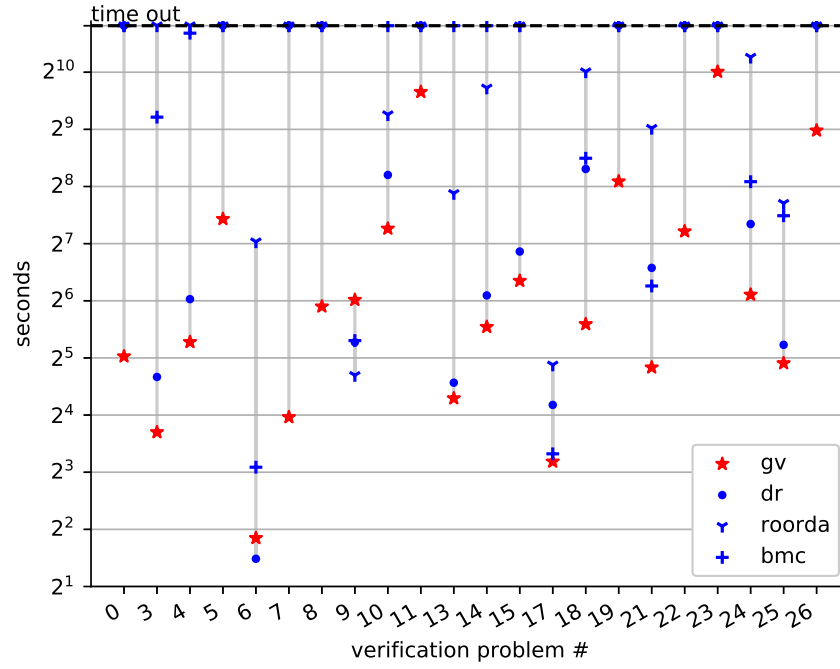


Figure 8.6: GV is faster than DR, Roorda and BMC at finding corner-case bugs

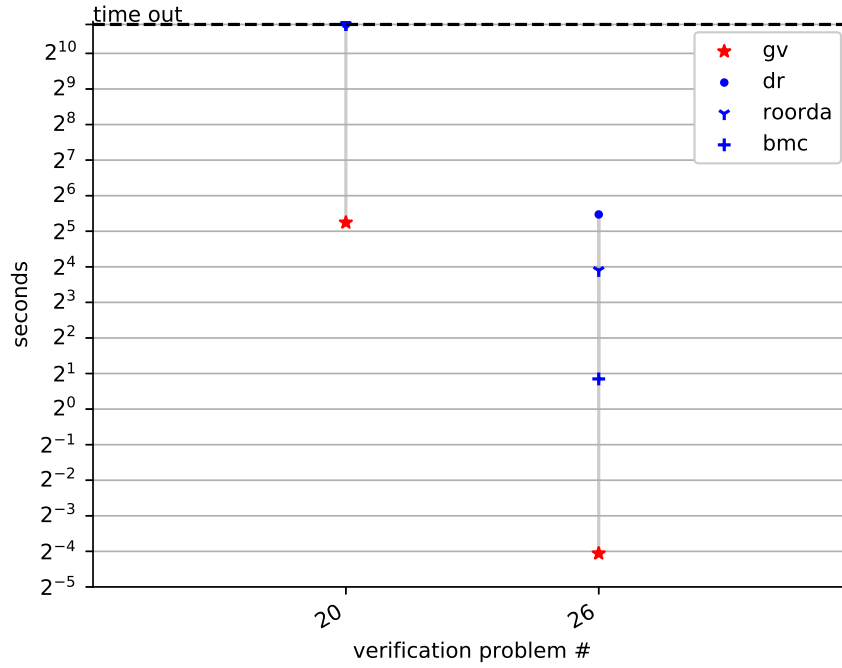


Figure 8.7: Effectiveness of GV memory abstraction

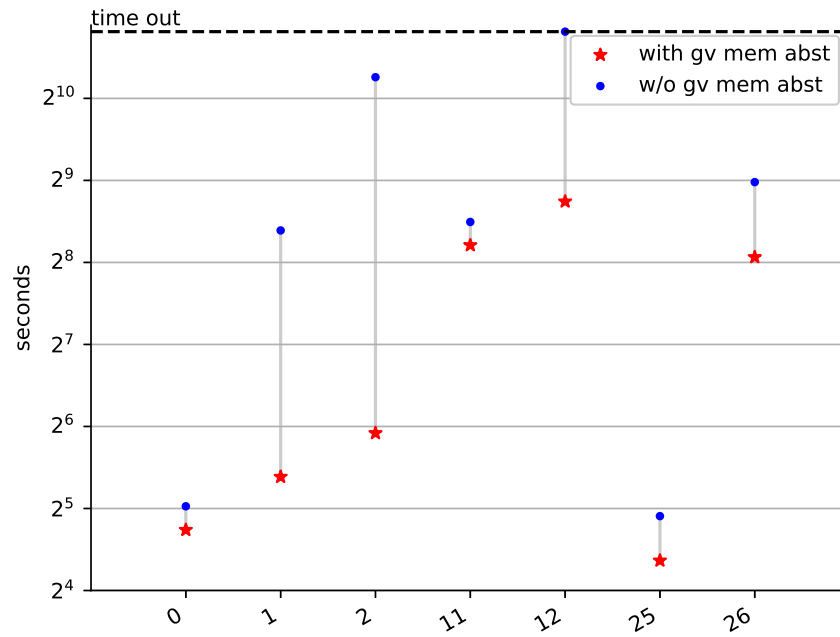


Figure 8.8: Effectiveness of GV bit-width reduction

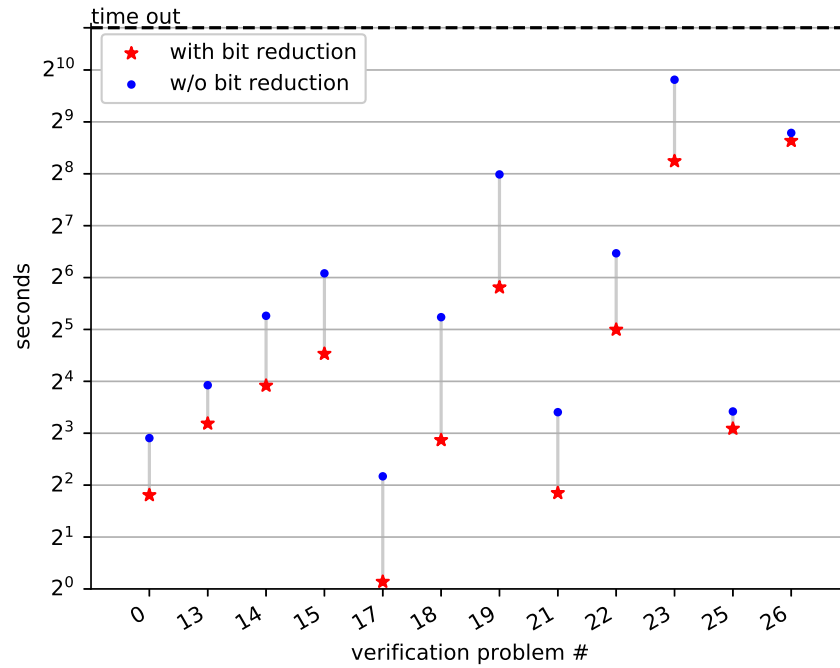
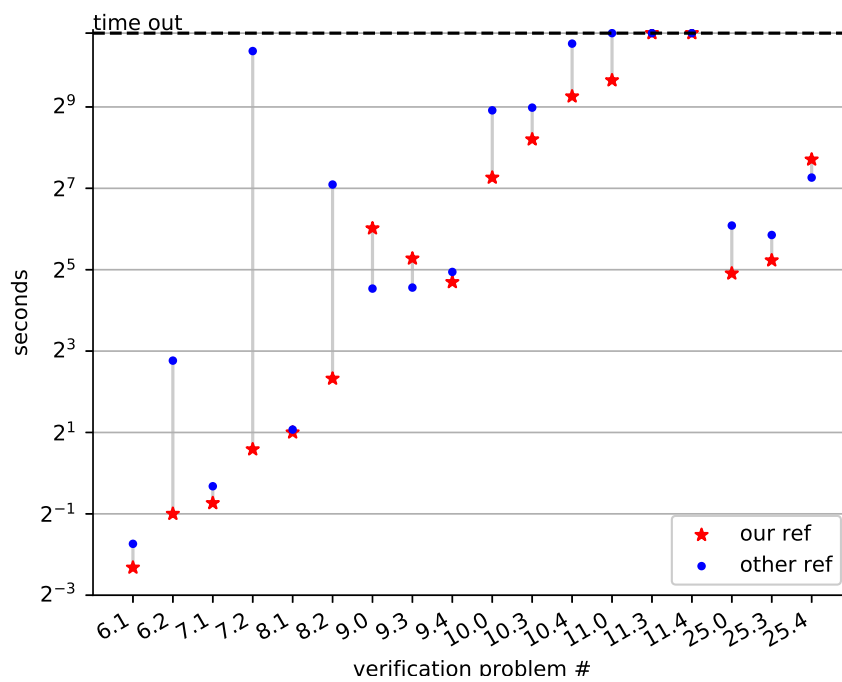


Figure 8.9: Effectiveness of ternary refinement



8.2.3 CAM

With the default parameters, CAM has 32 entries and each entry's tag is 8-bit wide (TWIDTH=8) and each entry's data is 2-bit wide (DW = 2). It takes 390 registers and 10870 AND gates to implement. The data correctness property is checked in cycle 18. The uniqueness property is inductive and proved by induction.

Data Correctness

Table 8.2 lists the number of clauses and verification time for various techniques. All techniques timed out except GV, which only needs 32.6 seconds. The number of clauses generated by Roorda being much larger than BMC indicates the overhead in encoding the data abstraction.

Table 8.2: CAM (data correctness): comparison of GV, DR, Roorda and BMC

	GV		DR		Roorda		BMC	
	v cl#	time	cl#	time	cl#	time	cl#	time
DW = 2	11585	32.6	53131	TO	276560	TO	172447	TO

Table 8.3 shows the results of applying GV memory abstraction. With GV memory abstraction, GV becomes much less sensitive to the increasing DW.

Table 8.3: CAM (data correctness): effective GV memory abstraction

	# of AND before simp		# of AND after simp		total time (seconds)
	value verif	guard verif	value verif	guard verif	
DW = 2	36219 (33975)	183765 (171701)	7129 (0)	0 (0)	32.6 (26.7)
DW = 16	62301 (55563)	318277 (302475)	35540 (0)	0 (0)	335.5 (41.8)
DW = 32	92109 (80235)	472005 (451931)	42425 (0)	0 (0)	1225.0 (60.5)

Results without memory abstraction are outside parentheses while results with memory abstraction are inside.

Table 8.4 shows that the bit-width reduction halves the verification time of GV. The bit-width reduction effectively reduces the tag width to 2.

Table 8.4: CAM (data correctness): effective bit-width reduction

	with bit-width reduction		without bit-width reduction	
	v AND before logic opt	vTime	v AND before logic opt#	vTime
DW = 2	10739	3.5	36219	7.5

Tag Uniqueness

Table 8.5 lists the number of clauses and verification time for various techniques. They are compared for increasing **TWIDTH**. GV easily outperforms the other techniques. In fact, GV can be done by just logic optimizations even for the largest **TWIDTH** we experimented with. The verification takes more time for Roorda than BMC stresses the importance of how to implement and utilize the data abstraction.

Table 8.5: CAM (tag uniqueness): comparison of GV, DR, Roorda and BMC

	GV		DR		Roorda		BMC	
	v cl#	time	cl#	time	cl#	time	cl#	time
TWIDTH = 8	0	13.0	0	25.4	20557	TO	17858	593.69
TWIDTH = 16	0	38.8	0	65.3	35513	TO	29803	1644.3
TWIDTH = 32	0	172.6	125665	TO	65437	TO	53491	TO

8.2.4 Scheduler

With the default parameters, the scheduler chooses the oldest ready process to execute from 16 ones. Each process' age is a 8-bit number. It costs 3214 AND gates and 128 registers to implement.

Table 8.6 lists the number of clauses and verification time for various techniques. They are compared for increasing AGE_WIDTH. As we can see, the value verification can be done using just the logic optimization even for the largest AGE_WIDTH. Because of the space limitation, we will use AW instead of AGE_WIDTH in the tables.

Table 8.6: Scheduler: comparison of GV, DR, Roorda and BMC

	GV		DR		Roorda		BMC	
	v cl#	time	cl#	time	cl#	time	cl#	time
AW = 8	0	3.6	6355	2.8	14264	131.0	4047	8.5
AW = 16	0	15.6	12734	TO	31207	TO	7973	TO
AW = 32	0	59.7	25586	TO	70510	TO	15866	TO

Recall that for our data abstraction for $lt()$ to work (i.e., not generating spurious CEXs), we need to refine it by modifying the implementation of $<$ (Section 7.4.1). Here, we introduce an alternative data abstraction, which is less abstracted but do not require modifying the implementation of $<$. Using such data abstraction is what existing refinement approach would do to refine our original data abstraction. The alternative data abstraction only uses X for the lower bits of the vector. For example, $\{\vec{a} : \vec{a} < 101\}$ is abstracted by $\{(0, X, X), (1, 0, 0)\}$ in the alternative data abstraction instead of $\{(0, X, X), (X, 0, 0)\}$.

We can abstract a process age with Xs when it is not ready. For both data abstractions, we have a choice between using $(\langle 0, 0 \rangle, \dots, \langle 0, 0 \rangle)$ or $(\langle 0, vage_{k-1} \rangle, \dots, \langle 0, vage_0 \rangle)$ for abstracting a process' age in this scenario, which are labelled with “encX 0” and “encX symb” respectively. Recall that $\vec{vage}$ is a vector of Boolean variables and is assumed to be the largest age among the entries of the scheduler. The 2 choices are equal if considered from the state space point of view. Table 8.7 compares the data abstraction using encX 0 and encX symb. We can see that when encX 0 is used, because of the value verification is too simple, our data abstraction is only slightly faster than the alternative data abstraction for the value verification. More importantly, the results show that our data abstraction is more robust because unlike the alternative data abstraction, less fluctuation in the verification time is observed for our data abstraction when using encX symb.

¹Using the Glucose SAT solver, it takes 1.4 seconds to solve for AGE_WIDTH = 32 and 1.1 seconds for AGE_WIDTH = 16 under the configuration of encX symb and alternative data abstraction. Therefore, the dramatic reduction (from 1326.4 s to 136.5) in the SAT time when the design parameter is increased is most likely to be due to the specifics of the miniSAT.

Table 8.7: Scheduler: comparison of the value verification using different combinations of data abstraction and encX 0 and encX symb

GV	data abst (alt data abst)					
	encX 0			encX symb		
	v AND	opt v AND	vTime	v AND	opt v AND	vTime
AW=8	3978(5592)	0(0)	0.2(0.3)	4234(5848)	0(2328)	0.5(6.8)
AW=16	8386(10940)	0(0)	0.6(0.8)	8898(11452)	0(4521)	1.5(1326.4)
AW=32	18178(21776)	0(0)	2.0(2.1)	19202(22800)	0(8819)	5.0 (136.5) ¹

Table 8.8 and Table 8.9 compare our refinement with existing refinement approaches that use the alternative data abstraction instead of modifying the design. When encX 0 is used, the existing refinement approaches are faster for GV because the value verification is much simpler than the guard verification in this case. Therefore, the reduction in time by our refinement for the value verification does not offset the increase in time by our refinement for the guard verification. When encX symb is used, the value verification becomes sufficiently more difficult than the guard verification for the existing refinement approaches which use the alternative data abstraction. Therefore, our refinement turns out better in this case. Existing refinement works better for DR and Roorda, likely because existing refinement’s data abstraction uses fewer gates and it leads to smaller verification problems. When $\text{AGE.WIDTH} \geq 16$, it always times out for DR and Roorda no matter which refinement method is used. Therefore, we did not record the size information.

Table 8.8: Scheduler: refinement comparison, encX 0

with existing ref (with our ref)		size before opt	size after opt	oTime	time
AW=8	GV	$\langle 22155, 5592 \rangle$ $(\langle 23045, 3978 \rangle)$	$\langle 0, 0 \rangle$ $(\langle 0, 0 \rangle)$	$\langle 1.9, 0.3 \rangle$ $(\langle 3.6, 0.2 \rangle)$	1.9 (3.8)
	DR	10784 (8707)	0 (3314)	1.6 (2.0)	1.6 (4.7)
	Roorda	11793 (14264)	-	-	60.3 (131.0)
AW=16	GV	$\langle 43103, 10940 \rangle$ $(\langle 50504, 8386 \rangle)$	$\langle 0, 0 \rangle$ $(\langle 0, 0 \rangle)$	$\langle 6.45, 0.8 \rangle$ $(\langle 15.31, 0.6 \rangle)$	7.3 (15.4)
	DR	- (-)	- (-)	- (-)	TO (TO)
	Roorda	- (-)	-	-	TO (TO)
AW=32	GV	$\langle 84854, 21776 \rangle$ $(\langle 111336, 18178 \rangle)$	$\langle 0, 0 \rangle$ $(\langle 0, 0 \rangle)$	$\langle 25.2, 2.2 \rangle$ $(\langle 57.7, 2.0 \rangle)$	27.4 (59.7)
	DR	- (-)	- (-)	- (-)	TO (TO)
	Roorda	- (-)	-	-	TO (TO)

Table 8.9: Scheduler: refinement comparison, encX symb

with existing ref (with our ref)		size before opt	size after opt	oTime	time
AW=8	GV	$\langle 22381, 5848 \rangle$ $((\langle 23285, 4234 \rangle))$	$\langle 0, 2328 \rangle$ $((\langle 0, 0 \rangle))$	$\langle 3.1, 1.0 \rangle$ $((\langle 5.6, 0.5 \rangle))$	10.0 (6.1)
	DR	11056 (8963)	0 (3549)	1.3 (2.0)	1.3 (8.6)
	Roorda	11809 (14264)	-	-	58.9 (108.0)
AW=16	GV	$\langle 43577, 11452 \rangle$ $((\langle 50984, 8898 \rangle))$	$\langle 0, 4521 \rangle$ $((\langle 0, 0 \rangle))$	$\langle 14.5, 1.9 \rangle$ $((\langle 24.6, 1.6 \rangle))$	1538.8 (26.1)
	DR	- (-)	- (-)	-(-)	TO (TO)
	Roorda	- (-)	-	-	TO (TO)
AW=32	GV	$\langle 85824, 22800 \rangle$ $((\langle 112296, 19202 \rangle))$	$\langle 0, 8819 \rangle$ $((\langle 0, 0 \rangle))$	$\langle 48.1, 4.6 \rangle$ $((\langle 151.2, 5.1 \rangle))$	200.4 (156.3)
	DR	- (-)	- (-)	-(-)	TO (TO)
	Roorda	- (-)	-	-	TO (TO)

8.2.5 Response Buffer

With the default parameters, RBuf has 17531 AND gates and 1423 REGISTERS. In the initial state, any request ID could be unavailable and any table entry could be used. When verifying RBuf, we write lemmas about controls signals being non-X to simplify the guard verification. The time used to verify the lemmas is added into the total verification time.

Data Correctness

Table 8.10 lists the number of clauses and the verification time for various techniques. They are compared for increasing bounds. GV always has smaller value verification than the others. The verification time for GV scales better than the others. Roorda is faster than BMC in the table while having more clauses demonstrates the power of the state space reduction.

Table 8.10: RBuf (data correctness): comparison of GV, DR, Roorda and BMC

data correctness	GV		DR		Roorda		BMC	
	v cl#	time	cl#	time	cl#	time	cl#	time
cycle 6	0	64.7	0	38.7	303058	25.9	167868	39.5
cycle 8	101144	153.5	110752	294.5	401948	612.4	222318	TO
cycle 10	144264	805.8	155191	TO	501009	TO	276768	TO

One may ask whether it is possible to adapt the idea of data abstraction to BMC. This way, we can get rid of the ternary logic and all the complexities associated with it, such as the extra guard verification, refinement and etc. In order to best implement the data abstraction in the Boolean domain for experiments here, for each subset that is created by partitioning the domain with predicates we choose to represent it with a symbolic value for every cycle and add constraints for the symbolic value. The constraints are the characteristic predicate for the subsets. In the RBuf example, the subsets are $\{\vec{v} : \vec{v} = \vec{a}\}$ and $\{\vec{v} : \vec{v} \neq \vec{a}\}$, where $\vec{a}$ is the arbitrarily picked request ID whose response data will be checked. To implement the data abstraction with BMC, we represent the subsets with $\vec{b}^i$ and $\vec{c}^i$ respectively for cycle i and add constraints that $\vec{b}^i = \vec{a}$ and $\vec{c}^i \neq \vec{a}$ for every cycle upto the bound. In Table 8.11, BMC-DA denotes BMC with the data abstraction. In this example, the data abstraction is shown to be beneficial though still significantly slower than GV and DR. However, we believe the improvement in the verification time is likely due to the special transformation that is done to the design as explained in Section 8.1.3 because in another example (Table 8.21), the data abstraction is a considerable overhead to BMC and dramatically harms its performance.

In Table 8.12, we implement a memory abstraction that is applicable to all of GV, DR, Roorda and BMC. This forms a competitive baseline to demonstrate the effectiveness of

Table 8.11: RBuf (data correctness): adapting the data abstraction to BMC

	BMC		BMC-DA	
	v cl#	time	cl#	time
cycle 6	167868	39.5	187158	18.1
cycle 8	222318	TO	247772	446.2
cycle 10	276768	TO	307972	TO

GV memory abstraction that is shown in Table 8.13. GV memory abstraction makes big differences for deeper bound and larger parameters. In Table 8.13, RBUF's number of table entries is increased to 32 from 8. GV memory abstraction increases the complexity of the guard verification (increased size) because of it introducing more $\mathbf{X}$, but as shown in Table 8.13 it is less significant than the improvement in the value verification.

Table 8.12: RBuf (data correctness): effective simple memory abstraction

	# of AND before simp		# of AND after simp		total time (seconds)
	value verif	guard verif	value verif	guard verif	
cycle 6	51414 (52028)	117246 (63027)	0 (0)	0 (0)	64.7 (25.7)
cycle 8	84166 (71746)	217740 (100448)	50911 (38171)	0 (0)	153.5 (66.3)
cycle 10	118198 (91464)	337194 (137874)	73555 (48856)	0 (0)	805.8 (131.5)

Results without memory abstraction are outside parentheses while results with memory abstraction are inside.

Table 8.13: RBuf (data correctness): effective GV memory abstraction

	# of AND before simp		# of AND after simp		total time (seconds)
	value verif	guard verif	value verif	guard verif	
cycle 10	118980 (129292)	185910 (228208)	58384 (0)	0 (0)	360.4 (296.1)
cycle 12	144354 (157292)	234704 (288896)	70924 (0)	0 (0)	TO (428.3)

Results without memory abstraction are outside parentheses while results with memory abstraction are inside.

Table 8.14 compares our refinement with existing ones. It shows that our refinement benefits all of GV, DR and Roorda except when the bound is shallow and the verification problem is too simple.

Table 8.14: RBuf (data correctness): refinement comparison

with existing ref (with our ref)		size before opt	size after opt	oTime	time
cycle 6	GV	$\langle 0, 43419 \rangle$ $\langle \langle 117246, 51414 \rangle \rangle$	$\langle 0, 26253 \rangle$ $\langle \langle 0, 0 \rangle \rangle$	$\langle 0, 17.9 \rangle$ $\langle \langle 34.8, 19.0 \rangle \rangle$	23.2 (64.7)
	DR	43419 (96667)	26253 (0)	18.2(38.7)	23.6 (38.7)
	Roorda	247561 (303058)	-	-	30.8 (25.9)
cycle 8	GV	$\langle 0, 71631 \rangle$ $\langle \langle 217740, 84166 \rangle \rangle$	$\langle 0, 46366 \rangle$ $\langle \langle 0, 50911 \rangle \rangle$	$\langle 0, 40.8 \rangle$ $\langle \langle 86.4, 46.5 \rangle \rangle$	483.2 (153.5)
	DR	71631 (160395)	46366 (56844)	41.6(115.4)	506.0 (294.5)
	Roorda	328471 (401948)	-	-	1505.7 (612.4)
cycle 10	GV	$\langle 0, 100659 \rangle$ $\langle \langle 337194, 118198 \rangle \rangle$	$\langle 0, 66351 \rangle$ $\langle \langle 0, 73555 \rangle \rangle$	$\langle 0, 85.2 \rangle$ $\langle \langle 198.2, 78.6 \rangle \rangle$	TO (805.8)
	DR	100659 (226427)	66351 (81322)	83.0(212.0)	TO (TO)
	Roorda	409392 (501009)	-	-	TO (TO)

ID Uniqueness

Table 8.15 lists the number of clauses and verification time for various techniques. They are compared for increasing bounds. As we can see, the value verification and DR can be done using just logic optimizations even for the deepest bound we tested. The logic optimizations for DR takes more time than GV. Roorda outperforms BMC because of the data abstraction.

Table 8.15: RBuf (ID uniqueness): comparison of GV, DR, Roorda and BMC

	STE-GV		STE-DR		Roorda		BMC	
	v cl#	time	cl#	time	cl#	time	cl#	time
cycle 6	0	19.6	0	23.7	169804	235.5	157730	TO
cycle 8	0	46.6	0	68.3	225220	846.4	214476	TO
cycle 10	0	81.5	0	116.3	280796	TO	271222	TO

Table 8.16 shows that the bit-width reduction enabled by GV and data abstraction is effective and helps the verification time scales better with the bound.

Table 8.16: RBuf (ID uniqueness): effective bit-width reduction

	with bit-width reduction		without bit-width reduction	
	v AND before logic opt	vTime	v AND before logic opt	vTime
cycle 6	41714	9.1	45648	15.2
cycle 8	59676	15.1	65038	38.4
cycle 10	77640	23.1	84430	67.7

8.2.6 Shared Resource Manager

With the default parameters, the shared resource manager has 32 distinct resources that are available to 2 consumers. It takes 19910 ANDs, 519 REGISTERS to implement.

Table 8.17 lists the number of clauses and the verification time for various techniques. Table 8.18 increases the design parameter that specifies the number of consumers from 2 to 4. Various techniques are compared for increasing bounds. Comparing with the others, the size reduction by data abstraction and logic optimizations in the value verification is significant. For DR, time-out happens during the logic optimizations. Thus, we were not able to provide the number of clauses in the tables for some bounds. Roorda is slower than BMC even with data abstraction and a smaller size, which demonstrates the difficulties in correctly using the data abstraction.

Table 8.17: SRM: comparison of GV, DR, Roorda and BMC

2 consumers	GV		DR		Roorda		BMC	
	v cl#	time	cl#	time	cl#	time	cl#	time
cycle 6	0	9.1	0	18.1	144021	29.4	165295	10.0
cycle 8	10021	48.2	22876	316.7	179655	1030.4	212555	360.6
cycle 10	18115	272.2	⁻²	TO	215300	TO	260473	TO

Table 8.18: SRM: comparison of GV, DR, Roorda and BMC

4 consumers	GV		DR		Roorda		BMC	
	v cl#	time	cl#	time	cl#	time	cl#	time
cycle 6	9705	28.5	20300	95.4	233881	519.2	268930	76.6
cycle 8	17909	148.5	⁻³	TO	292938	TO	346038	TO
cycle 10	27935	1029.3	⁻⁴	TO	351988	TO	422956	TO

Table 8.19 provides the results for using bit-width reduction when the number of consumers is 2 and 4. It shows that the bit-width reduction enabled by GV and data abstraction is very effective for verifying SRM and helps the verification time scales better with the bound.

In order to test GV's performance when a design has bugs. We introduce a bug which makes it possible for a consumer to acquire resources even if none is available. This bug can only occur after every resource is allocated which can happen only after 17 clock

²Simplification time-out, therefore no clauses number after simplification.

³Simplification time-out, therefore no clauses number after simplification.

⁴Simplification time-out, therefore no clauses number after simplification.

Table 8.19: SRM: effective bit-width reduction				
	with bit-width reduction		without bit-width reduction	
	v AND before logic opt	vTime	v AND before logic opt	vTime
cycle 6	21493 (31335)	1.1 (3.6)	34477 (51607)	4.5 (10.6)
cycle 8	26603 (39921)	7.3 (31.9)	48763 (75513)	37.7 (88.5)
cycle 10	31713 (48507)	56.1 (302.8)	63049 (99419)	253.7 (898.9)

Results with 2 consumers are outside parentheses while results with 4 consumers are inside.

cycles. Table 8.20 shows that GV can find the bug within 37.9 seconds while all other techniques time-out. Note that we did not apply logic optimizations for GV or DR because experiments show that they are usually detrimental for SAT solvers to find bugs.

Table 8.20: SRM: comparison of GV, DR, Roorda and BMC for buggy designs

2 consumers	GV		DR		Roorda		BMC	
	v cl#	time	cl#	time	cl#	time	cl#	time
cycle 18	61322	37.9	102637	TO	153093	TO	182705	TO

For SRM, as in Table 8.11, we also adapted data abstraction to BMC, which harms the performance in this case.

Table 8.21: SRM: adapting the data abstraction to BMC

2 consumers	BMC		BMC-DA	
	v cl#	time	cl#	time
cycle 6	165295	10.0	223561	28.7
cycle 8	212555	360.6	289221	1404.0
cycle 10	260473	TO	354464	TO

8.2.7 Floating Point Subtraction

With the default parameters, the floating point subtraction’s mantissa’s width is 512, the exponent’s width is 9 and the pipeline is 11 stages deep. It takes 54300 ANDs and 10895 REGISTERS to implement.

Table 8.22 lists the number of clauses and the verification time for various techniques. The property to verify is inductive. Both GV and DR can solve the verification problem purely by logic optimizations. But it takes less than half time of DR for GV. The number of clauses for Roorda is more than twice as many as BMC because of the modifications we did to the design to enable data abstraction described in Section 8.1.5.

Table 8.22: FPSub: comparison of GV, DR, Roorda and BMC

GV		DR		Roorda		BMC	
v cl#	time	cl#	time	cl#	time	cl#	time
0	68.9	0	162.4	256899	1229.6	96109	271.4

8.2.8 Bypass

With the default parameters, bypass has a 5-stage pipeline and each instruction read and write 64-bit data using 8-bit address. It takes 120787 AND gates and 17774 REGISTERS to implement. Because BMC cannot solve it even for the smallest bound we verify, we apply the “simple” memory abstraction to all of the techniques. The memory abstraction reduce the memory of 2^8 cells to just one cell. With the memory abstraction, it takes 22800 AND gates and 1454 REGISTERS to implement.

Table 8.23 lists the number of clauses and the verification time for various techniques. GV outperforms the others especially as we increase the bound.

Table 8.23: Bypass: comparison of GV, DR, Roorda and BMC

	GV		DR		Roorda		BMC	
	v cl#	time	cl#	time	cl#	time	cl#	time
Cycle 11	11109	30.0	20903	37.5	491868	209.0	359579	179.6
Cycle 15	29447	504.7	59807	TO	691760	TO	500111	TO

Table 8.24 provides the results for using bit-width reduction with GV. It clearly reduces the verification time even though not as much as in other examples.

Shown in Table 8.25, GV memory abstraction simplifies the value verification and makes bigger difference as the bound increases.

Table 8.24: Bypass: effective bit-width reduction

	with bit-width reduction		without bit-width reduction	
	v AND before logic opt	vTime	v AND before logic opt	vTime
cycle 11	12723	8.5	18421	10.7
cycle 15	26513	397.1	37099	441.8

Table 8.25: Bypass: effective GV memory abstraction

	# of AND before simp		# of AND after simp		total time (seconds)
	value verif	guard verif	value verif	guard verif	
cycle 11	18421 (24379)	106621 (131609)	5935 (0)	0 (0)	30.0 (20.6)
cycle 15	37099 (48190)	220664 (290852)	29447 (9655)	0 (0)	504.7 (267.6)

Results without memory abstraction are outside parentheses while results with memory abstraction are inside.

To demonstrate GV’s effectiveness for verifying designs with bugs. We insert a bug that results in the table not updating the number to add correctly. The bug occurs after 12 cycles. Again, we disabled logic optimizations for GV and DR. Results can be found in Table 8.26. GV outperforms the others in finding bugs in the bypass design in orders of magnitudes. This experiment also shows that GV memory abstraction also benefits bug hunting in the bypass.

Table 8.26: Bypass: comparison of GV, DR, Roorda and BMC for the buggy bypass

	GV		GV+mem abst		DR		Roorda		BMC	
	v cl#	time	v cl#	time	cl#	time	cl#	time	cl#	time
cycle 15	52236	0.25	67634	0.06	160383	44.4	691052	14.9	499845	1.8

Table 8.27 shows that our refinement approach benefits both of GV and DR for the bypass. It is not helpful for Roorda possibly because the performance of Roorda technique is more sensitive to the size of the verification problem.

Table 8.27: Bypass: refinement comparison

with existing ref (with our ref)	size before opt	size after opt	oTime	time
GV	$\langle 96140, 18025 \rangle$ $(\langle 106621, 18421 \rangle)$	$\langle 0, 11244 \rangle$ $(\langle 0, 5935 \rangle)$	$\langle 17.3, 20.0 \rangle$ $(\langle 18.6, 9.0 \rangle)$	67.9 (30.0)
DR	39470 (42018)	14426 (11087)	36.1(28.6)	57.8 (37.5)
Roorda	470422 (491868)	-	-	153.7 (209.0)

Chapter 9

Conclusion

In this thesis, we presented data abstraction and memory abstraction techniques that are based on ternary abstraction and only made possible by the guard-value encoding. Our data abstraction needs predicates to abstract states similar to predicate abstraction. But there are some key differences which make our data abstraction more attractive and practical to use for hardware verification. In our data abstraction, the predicates we use only partition the state space for a small number of selected signals while the other signals' state space remains the same. However, predicate abstraction uses a set of predicates to partition the entire state space. It is easier to come up with a set of predicates that captures enough information about the states to avoid spurious CEX for a subset of signals than for all of the signals. Besides, in predicate abstraction, for each cycle, to compute the reachable abstract states, predicate abstraction has to solve satisfiability problems, which could be expensive.

Besides data abstraction and memory abstraction, we presented distinctive refinement approaches for ternary abstraction that increase the precision of a ternary transition system by modifying the (unrolled) Boolean transition function instead of substituting X with fresh Boolean variables in the ternary stimulus and initial states. Existing approaches, which substitute variables, tend to significantly degrade the effectiveness of the ternary abstraction. We formalized each technique, presented theories, provided guidelines or templates for how to use it effectively and presented use cases that can be adapted for a wide range of designs and properties to verify. Our data abstraction and memory abstraction produce great synergy with off-the-shelf logic optimizations, i.e., applying logic simplifications to the AIGs before translating to CNF formulas greatly improves the time spent in SAT solving. We also showed how data abstraction can sometimes enable bit-width reduction that reduces a vector of signals to just 1 bit.

We demonstrated the effectiveness of data abstraction, the SAT-based implementation of STE using guard-value encoding and logic optimization, the guard-value memory abstraction and our refinement approach by experimenting with them on 6 complex designs ranging from a buffer design that is needed for processing traffic from real-world bus pro-

tocols to a synthetic design based on a superscalar microprocessor’s bypass path. In the results, we usually observed at least $2\times$ and often $10\times$ improvement for each of data abstraction and memory abstraction and more importantly, they can improve the scalability of the BMC with regard to the time bound as well as design parameters. It was also demonstrated that our value verification not only has smaller state space but also much smaller size in AIG representation than the corresponding Boolean verification. We also observed that our techniques work equally well for correct designs and designs with bugs. As for the refinement methods, it was observed that ours are better than the existing ones most of the time. For designs where bit-width reduction is possible because of our data abstraction, we observed that it consistently improved the verification time upto $4\times$. These results back up our theory of that our data abstraction and memory abstraction can significantly reduce the input state space and memory state space respectively in the value verification. The dramatic reduction in overall verification time is also owed to the fact that the reduction in the states space can be recognized by the logic optimizations to simplify the verification problem making it much smaller than BMC. Recall that comparing with other encodings, the guard-value encoding does not use the least number of Boolean gates to encode a ternary circuit. As a result, the size of the guard verification could be disconcerting on the surface, but the results showed that the guard verification problem is filled with logic redundancy because logic optimizations reduced it to a fraction of its original size and often even solved it directly. The improvement in overall verification time clearly tells that the cost of guard verification is not meaningful for complex designs and is more than offset by the time saved in value verification.

As future work, we plan to improve and extend our work. Improvement will include enhancing logic optimizations that we use to simplify verification problems generated by our abstraction techniques, and the automation of data abstraction, memory abstraction and refinement. Our logic optimizations are off-the-shelf and we have put little effort to tailor them for our need. Therefore, it is promising to study how to use the existing optimization algorithms more effectively or even creating new logic optimization algorithms that can better take advantage of the optimization opportunities provided by our data abstraction and memory abstraction. Automating these techniques is important to make them accessible to more users. Extending our work includes extending data abstraction to handle more general set of predicates and extending our techniques to IC3 ([9, 10]). Extending to IC3 is likely to be not straightforward. IC3 maintains a trace of set of states $(F_0, F_1, \dots, F_k)$, where F_i is an over-approximation of the states reachable within i steps. To use our techniques, F_i would be abstracted with $\hat{F}_i$ and represented as a vector of symbolic ternary values. As part of the algorithm of IC3, it is needed to check whether $\hat{F}_i$ is inductive, which poses challenges because of the nested quantifiers involved when $\hat{F}_i$ is represented as a vector of symbolic ternary values: checking whether $\hat{F}_i$ is inductive is to check that for any state generated by $\text{Tr}(\hat{F}_i, \hat{\Sigma}_I)$, where $\hat{\Sigma}_I$ is a stimulus that assigns inputs to symbolic values, there exists a state generated by $\hat{F}_i$ abstracting it.

References

- [1] Mark D Aagaard, Robert B Jones, and Carl-Johan H Serger. Formal verification using parametric representations of boolean constraints. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 402–407, 1999.
- [2] Sara Adams, Magnus Bjork, Tom Melham, and Carl-Johan Seger. Automatic abstraction in symbolic trajectory evaluation. In *Formal Methods in Computer Aided Design (FMCAD’07)*, pages 127–135. IEEE, 2007.
- [3] Sara Elisabeth Adams. Abstraction discovery and refinement for model checking by symbolic trajectory evaluation. 2014.
- [4] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 193–207. Springer, 1999.
- [5] Armin Biere, Tom van Dijk, and Keijo Heljanko. Hardware model checking competition 2017. In *2017 Formal Methods in Computer Aided Design (FMCAD)*, pages 9–9. IEEE, 2017.
- [6] Per Bjesse. Word-level sequential memory abstraction for model checking. In *2008 Formal Methods in Computer-Aided Design*, pages 1–9. IEEE, 2008.
- [7] Per Bjesse and Arne Borallv. Dag-aware circuit compression for formal verification. In *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 42–49. IEEE Computer Society, 2004.
- [8] Per Bjesse, Tim Leonard, and Abdel Mokkedem. Finding bugs in an alpha microprocessor using satisfiability solvers. In *International Conference on Computer Aided Verification*, pages 454–464. Springer, 2001.
- [9] Aaron R Bradley. Sat-based model checking without unrolling. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.
- [10] Aaron R Bradley. Understanding ic3. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 1–14. Springer, 2012.

- [11] Robert Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification*, pages 24–40. Springer, 2010.
- [12] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 1020 states and beyond. *Information and computation*, 98(2):142–170, 1992.
- [13] Supratik Chakraborty, Zurab Khasidashvili, Carl-Johan H Seger, Rajkumar Gajavelly, Tanmay Haldankar, Dinesh Chhatani, and Rakesh Mistry. Word-level symbolic trajectory evaluation. In *International Conference on Computer Aided Verification*, pages 128–143. Springer, 2015.
- [14] Pankaj Chauhan, Edmund Clarke, and Daniel Kroening. Experiments with sat-based symbolic simulation using reparameterization in the abstraction refinement framework. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 2004.
- [15] Pankaj Chauhan, Edmund M Clarke, and Daniel Kroening. A sat-based algorithm for reparameterization in symbolic simulation. In *Proceedings of the 41st annual Design Automation Conference*, pages 524–529. ACM, 2004.
- [16] Marsha Chechik, Benet Devereux, Steve Easterbrook, and Arie Gurfinkel. Multi-valued symbolic model-checking. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(4):371–408, 2003.
- [17] Hana Chockler, Orna Grumberg, and Avi Yadgar. Efficient automatic ste refinement using responsibility. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 233–248. Springer, 2008.
- [18] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.
- [19] Edmund M. Clarke, Reinhard Enders, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. *Formal methods in system design*, 9(1):77–104, 1996.
- [20] Edmund M Clarke, David E Long, and Kenneth L McMillan. Compositional model checking. 1989.
- [21] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In *International Conference on Computer Aided Verification*, pages 365–373. Springer, 1989.

- [22] Niklas Een, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, pages 125–134. FMCAD Inc, 2011.
- [23] Niklas Een, Alan Mishchenko, and Niklas Sörensson. Applying logic synthesis for speeding up sat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 272–286. Springer, 2007.
- [24] Niklas Een and Niklas Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.
- [25] E Allen Emerson and A Prasad Sistla. Symmetry and model checking. *Formal methods in system design*, 9(1):105–131, 1996.
- [26] Malay K Ganai and Adnan Aziz. Improved sat-based bounded reachability analysis. In *Proceedings of the 2002 Asia and South Pacific Design Automation Conference*, page 729. IEEE Computer Society, 2002.
- [27] Amit Goel and Randal E Bryant. Set manipulation with boolean functional vectors for symbolic reachability analysis. In *2003 Design, Automation and Test in Europe Conference and Exhibition*, pages 816–821. IEEE, 2003.
- [28] Amit Goel and Randal E Bryant. Symbolic simulation, model checking and abstraction with partially ordered boolean functional vectors. In *International Conference on Computer Aided Verification*, pages 255–267. Springer, 2004.
- [29] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with pvs. In *International Conference on Computer Aided Verification*, pages 72–83. Springer, 1997.
- [30] Arie Gurfinkel and Marsha Chechik. Multi-valued model checking via classical model checking. In *International Conference on Concurrency Theory*, pages 266–280. Springer, 2003.
- [31] C Norris Ip and David L Dill. Better verification through symmetry. In *Computer Hardware Description Languages and their Applications*, pages 97–111. Elsevier, 1993.
- [32] Andreas Kuehlmann. Dynamic transition relation simplification for bounded property checking. In *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.*, pages 50–57. IEEE, 2004.
- [33] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(12):1377–1394, 2002.

- [34] Kenneth L McMillan. Verification of an implementation of tomasulo’s algorithm by compositional model checking. In *International Conference on Computer Aided Verification*, pages 110–121. Springer, 1998.
- [35] Kenneth L McMillan. Interpolation and sat-based model checking. In *International Conference on Computer Aided Verification*, pages 1–13. Springer, 2003.
- [36] Thomas F Melham and Robert B Jones. Abstraction by symbolic indexing transformations. In *International Conference on Formal Methods in Computer-Aided Design*, pages 1–18. Springer, 2002.
- [37] Alan Mishchenko, Robert Brayton, Jie-Hong R Jiang, and Stephen Jang. Scalable don’t-care-based logic optimization and resynthesis. *ACM Transactions on Reconfigurable Technology and Systems (TRETs)*, 4(4):1–23, 2011.
- [38] Alan Mishchenko and Robert K Brayton. Sat-based complete don’t-care computation for network optimization. In *Design, Automation and Test in Europe*, pages 412–417. IEEE, 2005.
- [39] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. Dag-aware aig rewriting: A fresh look at combinational logic synthesis. In *2006 43rd ACM/IEEE Design Automation Conference*, pages 532–535. IEEE, 2006.
- [40] Alan Mishchenko, Satrajit Chatterjee, Robert Brayton, and Niklas Een. Improvements to combinational equivalence checking. In *2006 IEEE/ACM International Conference on Computer Aided Design*, pages 836–843. IEEE, 2006.
- [41] Alan Mishchenko, Satrajit Chatterjee, Roland Jiang, and Robert K Brayton. Fraigs: A unifying representation for logic synthesis and verification. Technical report, ERL Technical Report, 2005.
- [42] Manish Pandey, Richard Raimi, Randal E Bryant, and Magdy S Abadir. Formal verification of content addressable memories using symbolic trajectory evaluation. In *Proceedings of the 34th annual Design Automation Conference*, pages 167–172, 1997.
- [43] Jan-Willem Roorda and Koen Claessen. A new sat-based algorithm for symbolic trajectory evaluation. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 238–253. Springer, 2005.
- [44] Jan-Willem Roorda and Koen Claessen. Sat-based assistance in abstraction refinement for symbolic trajectory evaluation. In *International Conference on Computer Aided Verification*, pages 175–189. Springer, 2006.
- [45] Hamid Savoj. *Don’t cares in mult-level network optimization*. PhD thesis, University of California, Berkeley, 1992.

- [46] Carl-Johan H Seger and Randal E Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–189, 1995.
- [47] Niklas Sorensson and Niklas Een. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT*, 2005(53):1–2, 2005.
- [48] Rachel Tzoref and Orna Grumberg. Automatic refinement and vacuity detection for symbolic trajectory evaluation. In *International Conference on Computer Aided Verification*, pages 190–204. Springer, 2006.
- [49] Clifford Wolf. Symbiosys. <https://symbiosys.readthedocs.io/en/latest/index.html>. Accessed: 2021-08-04.
- [50] Clifford Wolf. Yosys open synthesis suite. <http://www.clifford.at/yosys/>. Accessed: 2021-08-04.
- [51] Qi Zhu, N Kitchen, A Kuehlmann, and A Sangiovanni-Vincentelli. Sat sweeping with local observability don’t-cares. In *2006 43rd ACM/IEEE Design Automation Conference*, pages 229–234. IEEE, 2006.

APPENDICES

Appendix A

Memory Abstraction Implementation for Benchmark Designs

A.1 CAM

```
1
2  logic [DWIDTH-1:0]  cam_data [DEPTH-1:0]; // data array
3  generate
4      for (I =0; I<DEPTH; I++) begin
5          always_ff @(posedge clk)
6              if (rst)
7                  cam_data[I] <= '0;
8              else if (wr_to_del[I] || wr_update[I] || wr_avail[I])
9                  cam_data[I] <= wdata;
10     end
11 endgenerate
```

Listing A.1: Without GV memory abstraction

```
1
2  generate
3      for (I =0; I<DEPTH; I++) begin
4          always_ff @(posedge clk) begin
5              if (rst)
6                  cam_data[I] <= '0;
7              else if (cared_entry[I]) begin
8                  if (wr_to_del[I] || wr_update[I] || wr_avail[I])
9                      cam_data[I] <= wdata;
10             end
11             else if ((wr_to_del | wr_update | wr_avail) &
12                     cared_entry )
```

```

13         cam_data[I] <= wdata;
14     end
15 end
16 endgenerate

```

Listing A.2: With GV memory abstraction

A.2 RBuf

```

1  // abstract this array
2  logic [2**ID_WIDTH-1:0] req_id_avail;
3  generate
4      for (I=0; I<2**ID_WIDTH; I++)
5          always_ff @(posedge clk)
6              if (insert && req_id == I)
7                  req_id_avail[I] <= 1'b0;
8              else if (resp_last && resp_id == I)
9                  req_id_avail[I] <= 1'b1;
10 endgenerate

```

Listing A.3: Without Boolean memory abstraction

```

1  logic verif_id_avail;
2  // rnd_req_id_avail is assigned fresh Boolean variables
3  // in the stimulus in every cycle
4  input [2**ID_WIDTH-1:0] rnd_req_id_avail;
5  // init_req_id_avail is assigned fresh Boolean variables
6  // in the stimulus in every cycle
7  input init_req_id_avail;
8  // verif_id is the arbitrarily picked ID to verify
9  logic [ID_WIDTH-1:0] verif_id;
10 always_ff @(posedge clk)
11     if (rst)
12         verif_id_avail <= init_req_id_avail;
13     else if (insert && req_id == I && I == verif_id)
14         verif_id_avail <= 1'b0;
15     else if (resp_last && resp_id == I && I == verif_id)
16         verif_id_avail <= 1'b1;
17
18 generate
19     for (I=0; I<2**ID_WIDTH; I++) begin
20         assign req_id_avail[I] = I == verif_id ? verif_id_avail :
21             rnd_req_id_avail[I];
22     end

```

23 `endgenerate`

Listing A.4: With Boolean memory abstraction

```

1      // abstract this array
2      logic [DATA_WIDTH-1:0] tbl_data [TBL_SIZE-1:0];
3      generate
4          for (I=0; I<TBL_SIZE; I++) begin
5              always_ff @(posedge clk)
6                  if (rst)
7                      tbl_data[I] <= '0;
8                  else
9                      if (resp_per_entry[I])
10                          tbl_data[I] <= {resp_data,
11                              tbl_data[I][DATA_WIDTH-1:RESP_DATA_WIDTH]};
12              end
13      endgenerate

```

Listing A.5: Without GV memory abstraction

```

1      generate
2          for (I=0; I<TBL_SIZE; I++) begin
3              always_ff @(posedge clk)
4                  if (rst)
5                      tbl_data[I] <= '0;
6                  else if (cared_entry[I]) begin
7                      if (resp_per_entry[I])
8                          tbl_data[I] <= {resp_data,
9                              tbl_data[I][DATA_WIDTH-1:RESP_DATA_WIDTH]};
10                  end
11                  else if (cared_entry & resp_per_entry)
12                      tbl_data[I] <= {resp_data,
13                          tbl_data[I][DATA_WIDTH-1:RESP_DATA_WIDTH]};
14      endgenerate

```

Listing A.6: With GV memory abstraction

A.3 Bypass

```

1      mem_dp #(.data_width (DATA_WIDTH),
2                   .addr_width (ADDR_WIDTH))
3      u_mem (.clock (clk),
4                   .wr_en (mem_wr_en),
5                   .wr_address (wr_mem_address),
6                   .rd_address (rd_mem_address),

```

```

7         .i_data (o_stage_val [NUM_STAGES-1]),
8         .o_data (mem_o_data));
9 // dual-port memory
10 module mem_dp (/*AUTOARG*/
11     // Outputs
12     o_data,
13     // Inputs
14     clock, wr_en, wr_address, rd_address, i_data
15 );
16 parameter data_width = 8;
17 parameter addr_width = 4;
18 input clock;
19 input wr_en;
20 input [addr_width-1:0] wr_address;
21 input [addr_width-1:0] rd_address;
22 input [data_width-1:0] i_data;
23 output [data_width-1:0] o_data;
24 logic [data_width-1:0] mem [2**addr_width-1:0];
25 logic [data_width-1:0] mem_output;
26
27 always_ff @(posedge clock) begin
28     if (wr_en)
29         mem[ wr_address ] <= i_data ;
30     mem_output <= rd_address == wr_address    && wr_en? i_data :
31                                                     mem[rd_address];
32 end
33
34 assign o_data = mem_output;
35
36
37 endmodule // mem

```

Listing A.7: Without Boolean memory abstraction

```

1 reg [DATA_WIDTH-1:0] mem_data;
2 // match_idx is the arbitrarily picked address to verify
3 logic [ADDR_WIDTH-1:0] match_idx;
4 // rnd_mem_output is assigned fresh Boolean variables
5 // in the stimulus in every cycle
6 input [DATA_WIDTH-1:0] rnd_mem_output;
7
8 // memory write
9 always_ff @(posedge clk) begin
10     if (reset)
11         mem_data <= init_data;
12     else if (mem_wr_en && (wr_mem_address == match_idx))

```

```

13     mem_data <= o_stage_val [NUM_STAGES-1];
14 end
15
16 // memory read
17 always_ff @(posedge clk) begin
18     if (mem_wr_en && (wr_mem_address == rd_mem_address) &&
19         (rd_mem_address == match_idx))
20         mem_o_data <= o_stage_val [NUM_STAGES-1];
21     else if (rd_mem_address == match_idx)
22         mem_o_data <= mem_data;
23     else
24         mem_o_data <= rnd_mem_output;
25
26 end

```

Listing A.8: With Boolean memory abstraction

```

1 // data array to abstract
2 logic [DATA_WIDTH-1:0] val_incr [NUM_STAGES-1:0];
3 generate
4     for (I=0; I<=NUM_STAGES-1; I++) begin
5         always_ff @(posedge clk) begin
6             if (reset)
7                 val_incr[I] <= 0;
8             else if (insert && valid_empty_entry[I])
9                 val_incr[I] <= init_val_incr;
10            else if (incr[I])
11                val_incr[I] <= val_incr[I] + incr_based_on_tag[I];
12            else if (reset_incr[I])
13                val_incr[I] <= incr_based_on_tag[I];
14        end
15    end
16 endgenerate

```

Listing A.9: Without GV memory abstraction

```

1 generate
2     for (I=0; I<=NUM_STAGES-1; I++) begin
3         always_ff @(posedge clk) begin
4             if (reset)
5                 val_incr[I] <= '0;
6             else if (cared_entry[I]) begin
7                 if (insert && valid_empty_entry[I])
8                     val_incr[I] <= init_val_incr;
9                 else if (incr[I])
10                    val_incr[I] <= val_incr[I] + incr_based_on_tag[I];
11                else if (reset_incr[I])

```

```

12         val_incr[I] <= incr_based_on_tag[I];
13     end
14     else if ((|(cared_entry & valid_empty_entry)) &&
15             insert)
16         val_incr[I] <= init_val_incr;
17     else if (cared_entry & incr)
18         val_incr[I] <= val_incr[I] +
19             incr_lkup_tbl[match_idx[2:0]];
20     else if (cared_entry & reset_incr)
21         val_incr[I] <= incr_lkup_tbl[match_idx[2:0]];
22     end
23
24     end
25 endgenerate

```

Listing A.10: With GV memory abstraction