From Finite to Infinite:
Scalable Automatic Verification of Hardware Designs and Distributed Protocols

by

Aman Goel

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2021

Doctoral Committee:

Professor Karem A. Sakallah, Chair
Professor John P. Hayes
Assistant Professor Manos Kapritsos
Professor Stéphane Lafortune

Aman Goel

amangoel@umich.edu

ORCID iD: 0000-0003-0520-8890

To my grandparents, Krishna & Shanti Prasad Goel, and my parents, Kavita & Bhushan Goel,
for their boundless love and support

ACKNOWLEDGMENTS

First and foremost, I would like to thank my adviser, Prof. Karem Sakallah, for giving me the support and guidance that I needed to succeed in my research. I have learnt everything from him, how to do research, how to think and explore new ideas, and most importantly, how to never give up and not be afraid of challenging research problems. The experience I gained working with him is invaluable, and this research work would not have been possible if not for the continued encouragement and support that Karem has provided.

I am also thankful to my dissertation committee: Prof. John Hayes, Prof. Manos Kapritsos, and Prof. Stéphane Lafortune for their support and constructive suggestions. Thanks are also due to my other amazing collaborators: Prof. Jean-Baptiste Jeannin and Prof. Baris Kasikci, as well as Denis, Katalin, Haojun, Hammad and Eli. I really enjoyed working with you all.

I got the opportunity to spend a summer at Cadence, Haifa, where I learnt a lot about industry-scale hardware model checking. Thanks to Ranan, Ziyad, and especially, Habeeb, for making that experience wonderful.

I am also thankful to the Yices team at SRI, especially Bruno, Dejan, Stephane, and Shankar, from whom I learnt a lot about automated reasoning techniques. Tools like Yices are the backbone of the model checking techniques I developed in my research, without which this work would not have been possible.

I am grateful to Ankush, Arun, Subarno, and all my friends that I made during my time at Michigan. They were always there to listen to my problems and provide me with the emotional support I needed during difficult times, as well as for making this experience so much fun.

Most importantly, I would like to express my sincere gratitude to my family: my grandmother, my parents, my brothers and sisters Vivek & Stuti, Sanit & Anchal, my nephews Samar & Sahir, and especially my life partner Dr. Sakshi, for being so amazing and supportive.

Finally, I thank the Computer Science & Engineering department and the University of Michigan for giving me this opportunity.

TABLE OF CONTENTS

CHAPTER

LIST OF FIGURES

FIGURE

LIST OF TABLES

TABLE

LIST OF APPENDICES

ABSTRACT

As the world increasingly depends on complex systems to transfer messages, store our data, control our finances, drive our cars, and manage our medical devices, how can we tell whether the system is correct, secure, and reliable? Common practices continue to employ computation-intensive simulation-based verification and tedious manual verification. Formal verification using model checking provides an automatic way to identify functional errors in human-engineered designs. State-of-the-art model checking techniques based on incremental induction, such as IC3/PDR, have gained significant success over prior approaches due to their property-directed nature and clever use of incremental satisfiability solving. However, bit-level IC3 can only be applied to synthesizable finite systems modeled as Boolean circuits. Moreover, even for finite hardware designs with fewer than a thousand state bits, bit-level IC3 struggles with scalability and becomes overwhelmed by low-level propositional learning as the state space of the problem increases.

In this dissertation, we present a collection of scalable techniques for automatically verifying word-level hardware designs and distributed protocols. We derive our intuition from the fact that the design-level representation of a system contains useful high-level information that can be utilized to offer full automation and better scalability. We first focused on hardware verification and developed a new way of separating what's important from uninteresting details using *equality abstraction* (EA), a novel technique that automatically infers the most crucial high-level information using equality relations over objects present in the word-level structural description of the problem, like branch conditions, operators, constants, and variables. We studied the intrinsic nature of these word-level designs, and combined EA with incremental induction at the word level to develop WIC3+EA, a fully-automatic word-level verification algorithm. We also developed a collection of customizable techniques, such as *hybrid data abstraction* and *proof race*, to complement WIC3+EA for scalable hardware verification. Our checker, called AVR, implements these procedures and has demonstrated significant benefits of these word-level verification techniques on a variety of open-source and industry designs, such as full-fledged RISC-V CPU cores, flash and video display controllers, and complex industrial-strength multiplier designs. AVR won the prestigious Hardware Model Checking Competition 2020, outperforming state-of-the-art model checkers with a wide margin.

Recognizing the lack of automation in the verification of parameterized systems, we then focused on automatically verifying infinite-state distributed protocols that contain an unbounded number of states. We developed automatic verification techniques that efficiently derive quantified inductive invariants required to establish the safety of unbounded distributed protocol specifications. This was accomplished by introducing simple extensions to the finite-domain incremental induction algorithm by taking advantage of three structural features in protocol specifications: a) the *spatial regularity* over unordered domains representing sets of identical replicas that can be permuted arbitrarily, b) the *temporal regularity* over totally-ordered domains that model the unbounded, but regular, evolution over time, and c) the *hierarchical* protocol structure. The key insight underlying our approach is that structural regularities and quantification are closely related concepts that express protocol invariance under different re-arrangements of its components or its evolution with time. Our approach is implemented in IC3PO, a new protocol verifier that significantly outperforms the state-of-the-art on a variety of distributed protocols, scales orders of magnitude faster, and robustly derives compact inductive invariants fully automatically. For instance, by carefully integrating these structural features, IC3PO was able to automatically prove the safety of Lamport's Paxos consensus protocol, generally viewed as a complex hard-to-understand algorithm. Notwithstanding its complexity, IC3PO inferred an inductive invariant for Paxos that identically matches the human-written one previously derived with significant manual effort using interactive theorem proving. While various attempts have been made to verify different versions of Paxos, to the best of our knowledge, we present the first demonstration of an automatically inferred inductive invariant for Lamport's original Paxos specification.

Interestingly, all our developed techniques provide provable correctness guarantees through automatic generation of mathematical explanations at the design level, referred as *certificates*, that accompany the verification result and can be checked independently to guarantee provable correctness. Our experiments showed that these techniques are successful in automatically verifying the correctness of several interesting complex systems, even ones whose subtleties and internal workings were unknown to us and were too hard to understand for a non-expert, with provable guarantees and assurance against critical design errors.

CHAPTER 1

Introduction

"*The heart of mathematics is abstraction – ignoring uninteresting details and finding what's important*" – Leslie Lamport[1]

Verification provides a way to identify functional errors in human-engineered designs. Automatic verification methods like model checking [91, 92] exploit the power of mathematics and automated logical reasoning techniques to check whether or not a design obeys a given specification. However, with the growing complexity of designs, achieving high scalability with full automation has become increasingly challenging, resulting in computationally expensive simulation-based verification or tedious manual verification through interactive theorem proving to be widely used in practice.

The primary goal of this dissertation is to explore algorithmic ways to scale automatic verification of complex systems—by incorporating often ignored high-level structural information available at the design level to identify interesting details and abstract away unimportant low-level specifics by using structure-driven abstraction techniques. Scaling automatic verification techniques to identify design errors, or to provably establish their absence, has a high practical value, and allows system developers to ensure correct functioning of safety-critical components. This, as a result, helps in identifying functional errors before deployment and in avoiding malicious attacks, down-times, and loss of revenue or reputation.

1.1 Preliminaries

The verification problem, in essence, boils down to checking whether or not a system obeys a given *property* under all possible executions. Formal verification using interactive theorem proving, e.g.,

[1] Leslie Lamport is a computer scientist best known for his seminal work in distributed systems and the document preparation system LaTeX. He won the 2013 Turing award for fundamental contributions to the theory and practice of distributed and concurrent systems.

[206, 102, 75], relies on tedious manual efforts and human expertise to derive a logic-driven answer to the verification problem. In contrast, model checking exploits the power of automated logical reasoning decision procedures, like satisfiability (SAT) solving [191, 192, 200], to check whether or not all possible system executions obey the specification by performing an algorithmic exploration of the design's state space. Model checking is one of the most successful verification techniques and allows checking the correctness of a system fully automatically, without the need for detailed understanding on the part of the user, and at the same time offers provable correctness guarantees. Model checking based on incremental induction, commonly referred as IC3 [58] / PDR [108],[2] has emerged as an effective automatic verification procedure that uses mathematical induction and logical reasoning to perform a property-directed approximate exploration of the design's state-space. IC3-style model checking, first introduced for bit-level verification of synthesized[3] hardware designs, has demonstrated significant success and adoption by the verification community both in academia and in industry [55].

1.2 The Problem

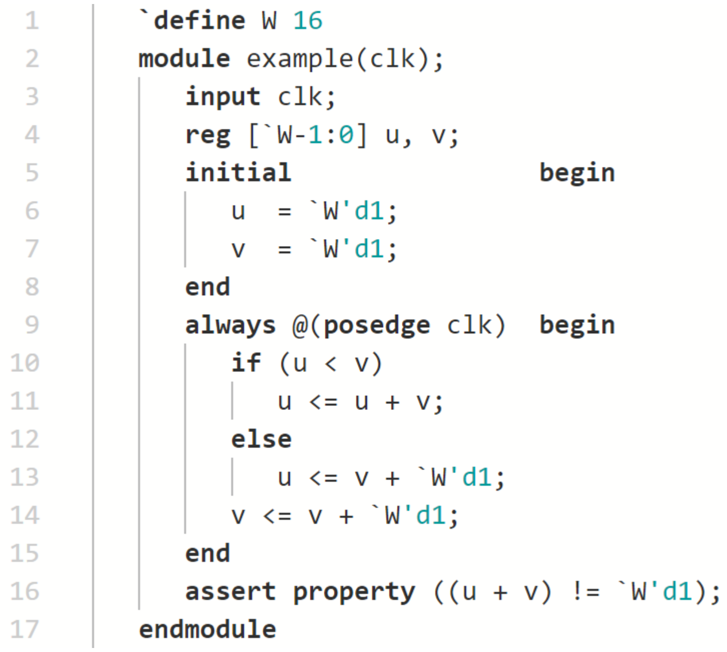
Given a design, generally represented as a symbolic state-transition system extracted from the source-code description, and a property as input, the model checking problem can be stated as follows: either prove that the property holds for all possible system executions or disprove the property by producing a counterexample. Our primary focus is on verifying *safety* properties, which basically assert that *nothing bad will ever happen*. A safety property is described as a formula that should hold on all states reachable from the design's initial state(s). For example, a traffic light controller may have a safety property that says traffic from crossing directions should never get green lights at once. In a distributed consensus protocol like Paxos [166, 167], a formula saying that at most a single value is chosen, is another example of a safety property. A safety property is referred to as an *invariant* if all possible system executions satisfy the safety property. An invariant Inv is *inductive* if it is closed under the system's transition relation, i.e., starting from a state inside Inv there is no way to escape to a state outside Inv. For safety property checking, the task of the verification tool, referred as *verifier* in short, is to check whether or not the safety property is an invariant of the given design. If the property can be violated (i.e., is not an invariant), the verifier should produce a counterexample witness, i.e., a valid execution trace of the system that ends up at an unsafe / bad state, i.e., a state that does not satisfy the property. The counterexample witness can then be used by the designer or an automatic debugger to identify the

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

[3] Hardware synthesis is the procedure of converting a word-level hardware description to a Boolean circuit consisting of basic logic gates.

source of the error and debug it. If the property holds (i.e., is an invariant), an IC3-style verifier can additionally produce an inductive invariant, which serves as a *proof certificate* that can be checked independently of the verifier to establish why the property is an invariant of the system.

Example 1: Simple Hardware Design



```
 1  `define W 16
 2  module example(clk);
 3     input clk;
 4     reg [`W-1:0] u, v;
 5     initial                begin
 6        u  = `W'd1;
 7        v  = `W'd1;
 8     end
 9     always @(posedge clk)  begin
10        if (u < v)
11           u <= u + v;
12        else
13           u <= v + `W'd1;
14        v <= v + `W'd1;
15     end
16     assert property ((u + v) != `W'd1);
17  endmodule
```

Figure 1.1: A simple hardware design in Verilog

Figure 1.1 presents a simple hardware design in Verilog [29, 227], described at the register-transfer level (RTL). This example encodes a finite state-transition system over two 16-bit state variables u and v (lines 1, 4). The initial state is defined as both u and v to be equal to 1 (lines 5-8), with the transition relation defined by lines 9-15. Line 16 describes the safety property asserting that the bitvector addition of u and v should not be equal to 1. The size of the state space is $2^{2\times 16}$ i.e., ~ 4.2 billion.

Example 2: Client Server Distributed Protocol

Figure 1.2 shows the description of a client server protocol, written in the Ivy language [210]. In this protocol, every server s maintains a lock, with the server's state represented with a Boolean *semaphore(s)* indicating whether it currently holds its lock. Every client-server pair is associated with a Boolean *link(c, s)* which denotes whether client c holds the lock of server s. Initially every server holds its own lock and all client-server links are set to false (lines 5-7). There are two possible actions in this protocol. A client may send a lock request to a server and acquire that

```
 1  type client
 2  type server

 3  relation semaphore(S:server)
 4  relation link(C:client, S:server)

 5  after init  {
 6      forall S.       semaphore(S)  := true;
 7      forall C, S.    link(C, S)    := false;
    }

 8  action connect(c:client, s:server) = {
 9      require semaphore(s);
10      link(c, s)    := true;
11      semaphore(s)  := false;
    }

12  action disconnect(c:client, s:server) = {
13      require link(c, s);
14      link(c, s)    := false;
15      semaphore(s)  := true;
    }

16  invariant [safety] forall C1, C2, S. link(C1, S) & link(C2, S) -> C1 = C2
```

Figure 1.2: Client server example in Ivy from [232, 210]

server's lock if the server currently holds its lock (lines 8-11). A client may also release a lock, handing it back to the server (lines 12-15). Line 16 describes the safety property of the protocol as "no two clients can have a link to (i.e., hold the lock of) the same server at the same time."

1.3 Challenges

Several theoretical and practical challenges limit applying automatic verification on real-world systems at scale. Here we present some of these bottlenecks relevant to this work.

Applicability: IC3 / PDR-based verification is arguably the most successful technique for model checking, especially for hardware. However, these techniques were proposed for finite-state systems modeled as a Boolean circuit and cannot be directly applied to word-level hardware designs or to distributed protocols composed of unbounded domains. As a result, common practice continues to employ computation-intensive simulation-based verification and expensive manual verification instead, while still leaving several important features of real-world designs unverified in practice.

Scalability: Even for finite-domain systems like synthesized hardware circuits, bit-level IC3 struggles with scalability due to being overwhelmed by bit-exact computations. As the size and complexity of the problem increases, the bit-level IC3 algorithm suffers from two main scalability issues: a) poor SAT solver performance, and b) getting overwhelmed by low-level propositional learning. Thus, while bit-level incremental induction has become a standard technique in many industrial settings, it still fails to scale beyond a few thousand state bits.

Undecidability: Generally speaking, due to more expressive though seemingly undecidable features, a sound and complete automatic verification technique is not always possible; typically, it may fail to prove or disprove a given property. This is because automated logical reasoning over undecidable features like non-deterministic pushdown automata, higher-order logics, or undecidable fragments of first-order logic are known to be theoretically undecidable. As a result, verification experts often limit their exploration to manual or semi-automatic verification techniques based on interactive theorem proving, which require a detailed understanding of the intricate inner workings of the design and entail significant manual effort to guide proof development. Notwithstanding the undecidability result of Apt and Kozen [35],[4] these theoretical limitations are often too pessimistic for a large class of practical, human-engineered designs, which include sufficient information in their *structure* that make them amenable to automatic verification.

Complexity: Problem descriptions often involve low-level implementation details that are unimportant when trying to check for certain types of properties. For example, when trying to determine if a hardware multiplier design correctly implements the control circuitry for two's complement multiplication, the multiplier's control behavior isn't affected significantly by the width of the operands. Low-level implementation details are often hard to reason using automatic decision procedures used by model checking techniques. For example, the bit-precise memory model of a microprocessor is too hard to reason at scale, and, in fact, exact reasoning involving each memory cell is not required to verify the correctness of several parts of the processor, like the control unit. Furthermore, for efficiency and other design choices, several unimportant specifics are often tightly mixed with important details critical to the property making manual attempts at identifying important aspects extremely challenging.

[4] An interesting aside is this quote by Ed Clarke in [86]: "In any case, it is clear that the claim in [35] regarding the infeasibility of automatically checking the correctness of programs with many processes is unduly pessimistic."

1.4 Proposed Solutions

Our primary objective is to investigate and develop techniques to mitigate different challenges that limit applying automatic verification to real-world scenarios. Our proposed solutions have a common theme—they utilize different *structural features* from the problem domain and design specifications to improve automatic verification through abstraction-based techniques and incremental induction.

1.4.1 Abstraction

The process of simplifying a problem by removing irrelevant details is called abstraction. Abstraction methods create an approximation of the system with the hope of making it more tractable for analysis. Automatic abstraction techniques avoid enumerative exploration of each exact system behavior and offer a way to deal with the complexity of real-world systems.

Equality Abstraction: In particular, we believe that the design-level structure of the system contains useful high-level information that can be exploited to better deal with complexity and offer better scalability. We developed a new way of separating what's important from uninteresting details in a hardware design using *equality abstraction* (EA), that allows automatic extraction of crucial high-level information directly from the word-level structure of the problem description. EA uses objects present in the word-level syntactic description of the problem like branch conditions, operators, constants, variables, etc. to implicitly represent a much-reduced abstract state-space using *equality relations* such that only equality and disequality among the variables are preserved regardless of their exact bit-precise assignments. For example, EA uses word-level terms $\{1, u, v, u < v, u + v, v + 1\}$ to define an abstract state space for the simple hardware design shown in Figure 1.1. The key characteristic of EA that makes it effective is that the abstract state space size is solely dependent on the number of terms in the design. So even when a variable domain increases from, say, 2-bits to 64-bits, the abstract space size doesn't get affected.

Hybrid Data Abstraction: Several previous works [63, 38, 34, 177] have demonstrated the advantages of abstracting away data using *equality with uninterpreted functions* (EUF). EUF abstraction is most appropriate for problems that are completely control-centric, i.e., where establishing correctness is independent of the data state. Many practical hardware verification problems, however, do require limited logical reasoning over certain data operations that affect the control flow of the design. To accommodate for such cases, we developed *hybrid data abstraction* that selectively applies EUF abstraction to only a subset of wide data operations while retaining the interpretation of the remaining operations. This abstraction is parameterized by a user-specified interpretation threshold W which can range from 1 to the largest bit width in the design. Reachability queries employ EUF logic for word-level variables whose width is larger than W and bit-vector (BV) logic for variables whose width is less than or equal to W. This provides for a range of data abstractions that enable successfully handling a diverse set of hardware verification problems from different sources.

Finite Analysis: Parameterized systems like distributed protocols are unique in the sense that they exhibit high regularity among different protocol objects. For example, in the client server protocol from Figure 1.2, where each server can only be connected to at most a single client at any time, the protocol behavior when client c_1 connects to server s is essentially very similar to when another client c_2 connects to s. Furthermore, as shown in previous works [215, 36, 237, 39, 202], these behaviors saturate after reaching a *cutoff* size. So even when the original protocol may have an unbounded number of clients and servers, we can infer a correctness proof, or identify a counterexample in a buggy protocol, by only analyzing a finite number of clients and servers. Using this insight, we explored how to algorithmically infer inductive invariants from finite protocol instances of increasing sizes until *finite convergence* is reached, after which protocol behaviors saturate and the inductive proof extends directly to the unbounded protocol. Analyzing finite instances instead of the full-fledged unbounded protocol allows focusing on relevant protocol behaviors in a bounded and *decidable* manner and enables extracting inductive proofs that naturally generalize to the unbounded distributed protocol.

1.4.2 Incremental Induction

Commonly referred to as IC3/PDR, incremental induction is a powerful technique that automatically solves the model checking problem by implicitly exploring the design's state space by cleverly over-approximating and refining a sequence of reachability frontiers at increasing distances from the initial state(s). It is arguably the most successful technique for model checking, especially for hardware. Over the last decade, bit-level verifiers using IC3-style analysis have shown exceptional performance in hardware model checking competitions (HWMCC) [68, 55].

Word-level IC3 with Equality Abstraction (wIC3+EA): Instead of traditional propositional-level IC3 using SAT solvers, we developed a fully-automatic hardware verification technique that combines incremental induction with equality abstraction to allow scalable verification directly at the *source-code* level. Coupled with a novel counterexample-guided abstraction-refinement (CEGAR) [159, 81] step to eliminate spurious abstract behaviors, we developed a word-level incremental induction algorithm based on equality abstraction (wIC3+EA) by utilizing the advances in automatic satisfiability checking of first-order logic formulas using *satisfiability-modulo-theories* (SMT) [44, 46] solvers. We explored algorithmic solutions to address challenges in combining incremental induction with EA, and, in the process, developed interesting scalable extensions to the WIC3+EA algorithm. Our experiments with WIC3+EA on real-world hardware designs showed very good potential and demonstrated much better scalability against existing state-of-the-art open-source and commercial hardware verifiers, especially for control-intensive problems.

Spatial and Temporal Regularity: Distributed protocols exhibit two important forms of structural regularity—*spatial* regularity over unordered sorts representing sets of identical "replicas" that can be permuted arbitrarily, and *temporal* regularity over totally-ordered sorts that model their unbounded, but regular, evolution "over time." For example, the constants in a sort representing a finite set of k identical processes are essentially indistinguishable copies that can be permuted arbitrarily without changing the protocol behavior. Furthermore, the states of a process that can perform a certain action repeatedly, but at ever-increasing times, can be viewed as being similar even though they have different time stamps. We establish a connection between protocol regularity and quantification, namely that they are complementary ways of expressing invariance. By utilizing this connection, we extended the finite-domain IC3/PDR incremental induction algorithm to *boost* its clause learning and "grow" a given clause φ to a set of clauses that accelerate the convergence of incremental induction, and, more importantly, makes it possible to algorithmically encode this set of clauses by a *single logically-equivalent* and *compact quantified clause* Φ. This enables systematically inferring the required inductive invariant by learning quantified formulas whose quantifier prefix is a mix of alternating universal and existential quantifier that is determined by the syntactic structure of the clauses and is the key to generalize the results of finite analysis to unbounded domains.

Formula Structure: Another structural feature that can help improve automatic verification of distributed protocols is the *formula structure* of the protocol's transition relation. A simple protocol may be described by a flat formula as a function of the protocol's present- and next-state variables. On the other hand, more complex protocols are typically specified by nested formulas which are functions of present-state, next-state as well as *auxiliary* non-state variables that correspond to sub-formulas of the transition relation. This is analogous to intermediate signals called *wires* in a hardware design that label logic gates or large function blocks, such as adders, multipliers, etc. By allowing such auxiliary variables to appear explicitly in clauses learned during incremental induction, we can *implicitly* incorporate the quantifiers used in the auxiliary variable definitions and automatically infer invariants with complex quantifier alternations.

Hierarchical Strengthening: Complex distributed protocols are often specified *hierarchically* as a sequence of related protocols at increasing levels of detail [168, 170]. *Voting* [173], for example, is the very high-level abstraction of *Paxos* [171], that formalizes the way Lamport first thought about the Paxos consensus algorithm [166, 167] without getting distracted by details introduced by having the processes communicate by messages. Hierarchical strengthening provides an automatic procedure to infer the required inductive invariant for hierarchically specified distributed protocols, by automatically verifying high-level abstractions first and using invariants of these higher-level abstractions as *strengthening assertions* to verify the detailed lower-level protocol. This, in turn, allows single-level automatic verification techniques based on incremental induction to scale to complex protocols like *Paxos*, by stepwise verifying higher-level abstractions first and using their auto-generated proofs to incrementally build the proof for the lower-level protocol.

1.5 Push-button Verification with Provable Correctness

The implementation of our proposed techniques led to the development of two fully-automatic verification tools: 1) AVR (acronym for ***A**bstractly **V**erifying **R**eachability*) for hardware verification, and 2) IC3PO (acronym for ***IC3** for **Pro**ving **P**r**o**tocol **Pro**perties*) for verifying distributed protocols.

AVR [121] is a push-button model checker developed, primarily, for verifying safety properties of hardware designs directly at the source-code level. Given a (finite) hardware design expressed in Verilog at register-transfer level, AVR, primarily, uses WIC3+EA to perform incremental induction directly on a first-order logic encoding of the transition relation using SMT solvers, resulting in a fully automatic and highly scalable word-level model checking procedure. AVR also implements a variety of complementary verification techniques, such as *hybrid data abstraction* and *proof race*, to increase its scalability, as well as useful utilities, such as design statistics and graphical visualizations, to provide high-level insights on the input design. AVR has demonstrated its scalability on very large open-source and industrial designs,[5] including several model checking problems defined on full-fledged RISC-V [230, 37] CPU cores. AVR was also independently evaluated to be the best verifier in the 11^{th} edition of the prestigious Hardware Model Checking Competition (HWMCC 2020) [52].

IC3PO [127] is a tool we developed for automatically verifying the safety of infinite-state distributed protocols. IC3PO extends the finite-domain incremental induction algorithm by integrating the knowledge of different structural features present in a protocol specification. IC3PO significantly outperforms the state-of-the-art in protocol verification, scales orders of magnitude

[5] In a proprietary industrial setting, AVR was reported to scale to hardware circuits with more than 65,000 state bits.

faster, and robustly derives compact quantified inductive invariants fully automatically. In particular, given the multi-level hierarchy of the Paxos specification, we used IC3PO to automatically infer an inductive invariant that identically matches the human-written one previously derived with significant manual effort using interactive theorem proving. While various attempts have been made to verify different versions of Paxos, to the best of our knowledge, our work presents the first demonstration of an automatically-inferred inductive invariant for Lamport's original Paxos specification.

Provable assurance on the verification outcome is guaranteed by both, AVR and IC3PO, using *independently-checkable* proof certificates and counterexample traces. Upon termination, both verifiers will either produce a *proof certificate*, in the form of a state formula representing an inductive invariant, if the safety property holds, or, a *counterexample trace* if it fails. In both cases, confidence in the verification output is achieved by using an external proof checker to independently confirm the correctness of the proof certificate or a trace simulator depicting the sequence of transitions leading to the failure.

1.6 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 describes the relevant background and reviews previous work relevant to this research. The next two chapters present our work on hardware verification. Chapter 3 presents equality abstraction and its integration in a word-level incremental induction algorithm. Chapter 4 describes different hardware verification techniques implemented in AVR, along with a detailed experimental analysis of these techniques on problems from HWMCC 2020. Chapters 5 & 6 describe our work on automatic verification of distributed protocols. Chapter 5 describes the key insights that enable inferring inductive invariants from finite protocol instances, with Chapter 6 elaborating them further in the context of verifying Lamport's Paxos protocol. Finally, Chapter 7 concludes this dissertation with directions for future work.

CHAPTER 2

Background and Related Work

In this chapter, we detail the relevant background in §2.1 and review prior work in §2.2 to put the research presented in this dissertation in proper context.

2.1 Background

2.1.1 Notation

Our setting is standard first-order logic with the notions of *sort*, *universe*, *signature*, and *structure* defined in the usual way [47] and we use the standard notions of theory, validity, satisfiability, and logical consequence. A *term* is a constant symbol, or an n-ary function symbol applied to n terms. An *atom* is $\top$, $\bot$ or an n-ary predicate symbol applied to n terms. A *literal* is an atom or its negation, a *cube* is a conjunction of literals, and a *clause* is a negation of a cube, i.e., a disjunction of literals. A *formula* is a literal or the application of logical connectives and/or quantifiers to formulas. We will refer to all terms with a non-Boolean range as *words* and refer to words with 0-arity as *ground* words. A *partition assignment* for a quantifier-free formula φ is defined as a set of partitions, one for each sort, dividing the terms in φ into equivalence classes. An interpretation $\mathcal{I}$ assigns a meaning to terms by means of a uniquely determined (total) mapping $[\![\_]\!]^{\mathcal{I}}$ of such terms into the universe of its structure. A model of a formula φ for an interpretation $\mathcal{I}$ is a structure that satisfies φ, i.e., $[\![\varphi]\!]^{\mathcal{I}} = \top$. For example, the interpretation for the theory of free sort and function symbols, call it $\mathcal{I}_P$, maps terms into the universe of partition assignments. The interpretation for the theory of bitvectors, call it $\mathcal{I}_B$, maps terms into a universe composed of bitvector assignments.

2.1.2 Problem Description

Given a state transition system, let X and X' denote a set of present- and corresponding next-state variables and, using a suitable logic, let formulas $Init(X)$, $T(X, X')$, $R(X)$, and $P(X)$ denote, respectively, the initial state(s), the transition relation, the reachable states, and the desired safety property. When clear from context we will drop the direct dependence on X and X', and write P' as an abbreviation for $P(X')$. We assume that, other than $R(X)$, these formulas can be easily derived from a description of a system in a suitable modeling language and that their sizes are linear in the size of the description. For example, in the hardware domain, our focus is on verifying word-level designs described in Verilog RTL, which can be encoded as *finite* state transition systems that are naturally expressed in the `QF_BV` theory of SMT-LIB [46].

A model checking (MC) problem $\mathcal{P}$ can be described by a 4-tuple $\langle X, \mathit{Init}, T, P\rangle$ and can be stated as follows: either prove that P holds for any sequence of executions starting from a state in Init, or disprove P by producing a counterexample. The most general framework for solving this problem is application of the *invariance rule* [215, 36]:

$$\begin{array}{lll} \text{a)} & \text{Initiation:} & \mathit{Init}(X) \rightarrow \mathit{Inv}(X) \\ \text{b)} & \text{Consecution:} & \mathit{Inv}(X) \wedge T(X, X') \rightarrow \mathit{Inv}(X') \\ \text{c)} & \text{Safety:} & \mathit{Inv}(X) \rightarrow P(X) \\ \hline & \text{Conclusion:} & \Box\, P(X) \end{array} \tag{2.1}$$

which requires that we come up with an *auxiliary* formula $\mathit{Inv}(X)$ that a) is satisfied by the initial state(s), b) is closed under the transition relation, and c) implies the property. A formula that satisfies the first two requirements is an *inductive invariant* allowing us to re-phrase the goal of verification as *deriving an inductive invariant that implies the desired safety property*. Note that the inductive invariant required to complete the proof must satisfy $R \rightarrow \mathit{Inv} \rightarrow P$. Thus, this framework admits two special cases, namely $\mathit{Inv} = R$ and $\mathit{Inv} = P$ that deserve a brief mention. In the first case, the auxiliary formula is the smallest inductive invariant, namely the reachable-states predicate R. Assuming that it is possible to compute and express R efficiently, we show that P holds by proving that $R \wedge \neg P$ is unsatisfiable. In the second case, we show that P holds by proving that both $\mathit{Init} \wedge \neg P$ and $P \wedge T \wedge \neg P'$ are unsatisfiable. In general, though, it is either infeasible or intractable to compute R and unlikely that P is already inductive. Thus, completing a safety proof almost always requires identifying a formula Inv that satisfies (2.1) and is neither R nor P. It should also be noted that, in general, (2.1) does not have a unique solution leading to the secondary consideration of obtaining an *explainable*, and not just any, proof. Easy-to-understand proofs tend to be short, reflect the structural features of the transition system, involve only those variables that are necessary to establish the property being checked, and can be understood without much effort (e.g., by inspection).

2.1.3 Incremental Induction

A major milestone in model checking was the introduction of incremental induction [58, 108]. The idea was to employ fast incremental SAT solving to derive the auxiliary formula Inv in the invariance rule. In contrast to the earlier SAT-based model checking approach [54, 221, 82, 100] that unrolled the transition relation T and that was very effective at uncovering "shallow" counterexamples, IC3/PDR operated on a single copy of T and produced an independently-checkable inductive invariant or a counterexample. This is accomplished by a systematic procedure for incrementally learning clauses at different depths from the initial state in search of a *strengthening assertion* A that makes $A \wedge P$ an inductive invariant. Finding such an assertion proves that P holds; otherwise, a counterexample to induction (CTI) showing how P is violated is produced. The original idea was described in [59, 58] and implemented in the IC3 tool. Een et al. [108] re-implemented the algorithm, with several enhancements, and dubbed the approach and corresponding implementation property-directed reachability or PDR. In the sequel, we will refer to this approach as reachability by *incremental induction* or simply IC3/PDR. Here, we will cover some of the salient features of incremental induction that facilitate the discussion on our proposed extensions.

Let R_k represent the set of states reachable from the initial states Init within k steps ($k \geq 0$). The property holds if $R_\infty \subseteq P$. Finding the exact set of reachable states is intractable in practice. Instead, IC3 iteratively derives over-approximations of R_k, referred to as frames F_k, to represent the *approximate* set of states reachable from Init in at most k steps.

The computational core in IC3 involves solving the satisfiability of queries that ask if it is possible to reach a state in the set $\mathit{Dest}(X)$ from any state inside the set $\mathit{Src}(X)$ in just one step. Mathematically, this means checking if the formula [$\mathit{Src}(X) \wedge T(X, X') \wedge \mathit{Dest}(X')$] is satisfiable or not. Using these 1-step solver queries with different Src and Dest formulas, the algorithm iteratively refines the definition of the frames F_k until—either a counterexample trace from Init to $\neg P$ is derived (disproving P), or a frame becomes inductive i.e., $F_i = F_{i+1}$ (proving P). In the latter case, the converged frame F_i is in fact an inductive invariant Inv, expressible as $\mathit{Inv} = A \wedge P$, that establishes why P is an invariant for the system.

2.2 Related Work

Our work is best understood by taking a brief look back at the 40+ years of research in automated verification. This review is necessarily incomplete but does cover the key milestones that our research builds on.

2.2.1 Finite-state Model Checking

The quest for automating the verification of *concurrent* systems goes back to the late 1970s and early 1980s [85]. While the initial concerns (and arguments!) in model checking were about devising suitable temporal logics that can express complex liveness peoperties,[1] the immediate practical issue that needed to be addressed was managing the so-called "state-space explosion problem" [89, 86, 83]. The complexity of the original EMC model checker was linear in the number of states since it performed an explicit traversal of the state graph and was only applicable to systems with no more than a few thousand states [87, 88]. Scaling explicit model checking to a parameterized system consisting of n identical m-state processes was clearly infeasible since m^n quickly becomes too large for relatively small values of n.

This explosion, however, is not limited to parameterized systems: the state space of a hardware design with n state bits (flip-flops) is also exponential in n. One can argue that the adoption of model checking as a key technology for the verification of hardware, and to a lesser extent software, was largely due to the impressive advances in automated reasoning technology, such as Binary Decision Diagrams (BDDs) [61], Conflict-Driven Clause-Learning SAT solving (CDCL SAT) [190, 192, 200], and SAT Modulo Theories (SMT) [44, 99] over the last four decades.

Explicit MC: The EMC model checker [87], developed in the early 1980s to verify properties of concurrent programs, was based on an explicit representation of the state transition system; it computed R_∞ by enumerating the reachable states and checking if any violated P. EMC was able to handle up to about 10^5 state or roughly 17 state bits.

Symbolic MC: The 1980s and 1990s saw significant scaling of MC to much larger transition systems, primarily in the hardware domain, by employing newly developed automated reasoning tools. BDDs allowed for compact symbolic representations of sets of states by characteristic Boolean functions. Symbolic MC represented R_∞ by a BDD using a least-fixed-point forward image computation and achieved scaling to about 10^{20} states or roughly 66 state bits [64].

SAT-based Bounded MC: The development of modern SAT solvers in the mid-1990s [191, 192, 200] provided another opportunity to scale model checkers to design sizes larger than what BDD-based checkers could handle. The first attempt simply unrolled the transition relation k times and created a large SAT formula whose satisfiability implied the existence of a k-step counterexample. Dubbed bounded model checking (BMC) [54], this approach was very effective in uncovering shallow bugs; proving the absence of bugs, however, required unrolling the transition relation to its

[1] Our focus in this dissertation is on safety properties. This review of related work excludes discussion of approaches for the verification of liveness properties.

sequential depth which was infeasible. BMC extended the range of hardware designs that could be handled to those containing several hundred state bits and relatively short counterexamples [82] and is now routinely deployed in industry [133] and anecdotally finds counterexample traces consisting of hundreds of steps.

SAT-based Incremental Induction: The latest development to address the *state-explosion problem* in MC was a clever deployment of SAT solving to perform the IC3 algorithm. Unlike BMC, this approach uses 1-step SAT queries that do not require the unrolling of the transition relation and is currently the state-of-the-art in MC. Incremental induction was widely adopted for both hardware and software verification. The original idea was described in [59, 58] and re-implemented in [108] with several enhancements.

2.2.2 Abstraction Techniques

In parallel with the above advances in MC approaches, an orthogonal attack on complexity was based on approximating the underlying transition system using abstraction techniques. Coupled with a refinement step to eliminate spurious abstract counterexamples, this approach has come to be known as **C**ounter**E**xample-**G**uided **A**bstraction **R**efinement, or CEGAR [81], and is now a standard framework for both hardware and software verification. A variety of abstractions have been proposed within this framework, including the theory of abstract interpretation [96], predicate abstraction [132, 93] popularized by the Microsoft SLAM project [40, 41], lazy abstraction [140], visible and invisible abstraction [84].

2.2.3 IC3-based Extensions

Several approaches from different domains have been suggested to extend the bit-level IC3 procedure, generally by combining IC3 with a CEGAR procedure. In the hardware domain, the authors of [229] suggest *lazy abstraction* using invisible variables. The authors of [141, 142, 143] suggest a variety of techniques to further extend bit-level IC3, by abstracting away certain expensive data operations, using unconstrained new primary inputs to ignore parts of the system, and using *localization abstraction* to cut away irrelevant logic. All these approaches [229, 141, 142, 143] use bit-level IC3 as the core engine and suffer with the same scalability issues as with bit-level IC3.

Certain approaches propose performing word-level IC3 using SMT solvers. Ref. [231] generalized IC3 to the theory of bitvectors by using *polytopes* and interval simulation. The procedure is however limited in expressiveness to invariants that can be expressed compactly as a disjunction of polytopes. Beyond hardware, different approaches have been proposed to lift the IC3 procedure to richer logics and infinite state systems [77, 144, 56, 175, 57]. IC3ia introduced in [79, 80] uses *implicit predicate abstraction* to perform a word-level IC3 procedure and refines the abstraction by adding new predicates. CTIGAR [56] proposes using *lifting* solver queries to eliminate non-essential specifics while guaranteeing continuity.

Datapath abstraction has been applied for both hardware [63, 34, 177, 176] and software [38, 62] to check for control-centric properties by replacing wide datapath signals and associated operators with uninterpreted terms and functions leaving control state at the bit level and extends IC3-style reachability to the quantifier-free logic of equality with uninterpreted functions (EUF). The Averroes system [177, 176] demonstrated how data abstraction can be exploited to perform word-level IC3 on control-centric hardware designs. Averroes applies data abstraction coupled with a CEGAR data refinement step and was shown to scale verification to certain large industrial hardware designs. However, when the property being checked is dependent on data operations, for which data abstraction is ill-suited, Averroes doesn't scale and leads to an excessive number of data refinement iterations to repair the abstraction.

2.2.4 Verification of Parameterized Systems

Efforts in the verification community to add automation to the analysis of distributed protocols represented as parameterized systems consisting of an arbitrary number of identical processes date back to the late 1980s and early 1990s. Introduced by Lamport and based on temporal logic, the TLA+ language [168] is a way to specify and verify concurrent systems and distributed protocols and is widely used in industry. The TLA+ toolbox [22] provides the TLC model checker, which is primarily used as a debugging tool for verifying small finite protocol instances [236], and not as a tool for inferring inductive invariants.

From n to one or two: The intuitive notion of verifying a large (but finite) parameterized system of n identical processes by reducing it to a smaller system "with one or two processes" began to emerge in the mid-1980s [89]. This was later formalized by similar notions referred to as a *closure process* [86], an *'invariant' process* [160], and a *network invariant* [234]. In modern nomenclature, these are over-approximations of the reachable space of an n-process system that are independent of n and that can be used in a finite structural induction proof "to draw conclusions on a system of arbitrary size n from one with a fixed size m" [160]. In all cases, these invariants had to be provided by the user for the induction check to be performed.

Symmetry: An alternative attack on the state explosion problem was introduced in the Murφ verifier [146, 207]. Noting that the exponentially-sized state space of a system composed of n identical processes was highly symmetric, the Murφ language was extended with a *scalarset* type whose elements acted as fully-permutable indices that were used to capture the system's *structural* symmetries. These structural symmetries were then used to explicitly construct a *symmetry-reduced* reachable set which was significantly smaller than the full reachable set for n processes and still allowed verification of the n-process system without compromising soundness. Symmetry was further explored in [90, 109, 110, 222] to reduce the verification of an n-process system to that of a smaller *quotient* system at a small *cutoff* size, and to extend the system model to allow not just one but different sets of symmetric processes.

From one arbitrary n to all n: It's important to note at this point that all these approaches were basically finite-state verification schemes for a given *fixed* system size n. The invariants used in their proofs (whether provided by the user or automatically generated by symmetry reduction) were finite structures whose inductiveness for that size was established using quantifier-free reasoning. Interest in deriving *quantified* invariants that can be used to prove safety properties for *unbounded* protocols started to appear in the early 2000s. The pivot may have been the observation, by the developers of Murφ, that the size of the symmetry-reduced reachable set becomes constant for values of n larger than a given number, a phenomenon they dubbed "data saturation," which allows the verification of unbounded systems, i.e., systems with any number of processes.

Invisible Invariants: The first attempts at automatically deriving quantified invariants were reported in [215, 36]. Referred to as *invisible invariants* (because they were not supposed to be seen), they were "divined" using BDDs in a 3-step procedure: a) find the reachable states $R(N)$ of a sufficiently large, but finite, N, b) *project* $R(N)$ onto m out of the N processes for a small fixed value m, c) generalize the projection to any arbitrary set of m processes by universally quantifying over m process indices yielding the candidate invariant $\varphi \triangleq \forall i_1, \cdots, i_m : \psi(i_1, \cdots, i_m)$. To determine if the proposed φ is inductive and implies safety, it is checked against a system with N_0 processes where N_0 is related to the number of local states in each process. The intuition underlying this method was the assumption that the system is "sufficiently symmetric", and that its behavior can be captured by any m-subset of its processes. However, the heuristic choice of φ as a universally-quantified assertion over m processes was not guaranteed to be inductive or to imply the safety property. Still, the success of employing finite reasoning to infer assertions about infinite systems cannot be over-stated: rather than use symmetry to manage the state explosion, this approach leveraged symmetry to learn facts about unbounded systems.

Theorem Proving Approaches: The derivation of inductive invariants for practical distributed protocols continues to be mostly carried out through *refinement proofs* [30, 162, 164, 116] using interactive theorem provers, such as TLAPS [75, 95], Isabelle/HOL [206], Coq [102], Dafny [178], and Ivy [210]. The TLAPS proof assistant [75, 95] allows checking proofs manually written in TLA+, and has been used to verify several distributed protocols, including variants of Paxos [170, 73]. The approaches in [138, 232, 198, 156] are examples of manually-derived refinement proofs that show how a low-level implementation refines a high-level specification. IronFleet [138] uses a combination of refinement and reduction [182] and divides the refinement proof into two parts: first proving that the implementation refines an abstract protocol, and then proving that the abstract protocol refines the specification. Verdi [232], on the other hand, uses a series of system transformers. However, unlike model checking, all these methods require a detailed understanding of the intricate inner workings of the protocol and entail significant manual effort to guide proof development.

Unbounded Reasoning in the EPR Fragment: Many recent attempts at adding automation to the verification of distributed protocols leverage the ability of SMT solvers to reason in the *effectively propositional* (EPR) fragment of first-order logic, an extension of the Bernays-Schönfinkel-Ramsey class, for which checking satisfiability is decidable [179, 214]. The Ivy language and verifier [210, 197] were introduced to further efforts in automating the verification of unbounded protocols. States in Ivy are defined as relations or functions over uninterpreted types which are used to encode computational processes (e.g., server, client, proposer, acceptor, etc.) or unbounded data (e.g., time, epoch, round, key, etc.), whereas transitions are expressed as *actions* over these types. Given an Ivy encoding of a distributed protocol along with a quantified safety property, the Ivy verifier either proves that the property holds inductively or produces a graphical representation of a counterexample to induction (CTI) as a small finite model which the user is expected to analyze manually in order to generalize and eliminate it. This process is repeated until no more CTIs are generated and could entail significant manual effort. Padon et al. [211] used Ivy to manually refine CTIs and obtain an inductive invariant for a simplified version of Paxos in the decidable EPR fragment of first-order logic.

Recent Attempts: Notwithstanding the undecidability result of Apt and Kozen [35], many efforts to *automatically* infer quantified inductive invariants have been reported with the pace increasing in recent years. Verification of parameterized systems using SMT solvers is explored in MCMT [217], Cubicle [94], and paraVerifier [180]. Abdulla et al. [31] proposed *view abstraction* to compute the reachable set for finite instances using forward reachability until cutoff is reached. Dooley and Somenzi proposed FORHULL-N [104] to verify parameterized reactive systems by running bit-level IC3 over small finite instances and generalizing the learnt clauses into candidate universally-quantified proofs through a process of proof saturation and convex hull computation. These candidate proofs involve modular linear arithmetic constraints as antecedents in a way such that they approximate the system behavior beyond the current finite instance, and their correctness is validated by checking them until the cutoff is reached.

Alternatively, verification can be carried out using recent approaches such as UPDR [152], QUIC3 [135], Phase-UPDR [113], and fol-ic3 [154], that extend IC3/PDR to automatically infer quantified inductive invariants. UPDR [152] uses *diagram-based abstraction* to infer universally-quantified inductive invariants. Phase-UPDR [113], relies on additional annotations from the user in the form of *phase structures* to help guide the search. Noting that some protocols may require invariants that involve both universal and existential quantifiers, the next step in automatic invariant inference was based on the use of quantified separators which are formulas in prenex form with a fixed bound on the number of quantifiers such that each is either universal or existential. The problem is now cast, in the fol-ic3 verifier [153], as an enumerative search in the space of such separators with the goal of finding an inductive subset that implies the safety property. Similar ideas along these lines, namely searching in the space of bounded quantified formulas, were recently reported in [136, 235]. SWISS [136] performs an enumerative template-based search in an optimized and bounded invariant search space by checking candidate formulas with SMT queries. DistAI [235] performs a data-driven methodology using data samples from protocol simulation to enumerate possible candidate formulas, followed by checking them through SMT solving. If any of these candidate invariants fail, monotonic weakening of the invariant set is performed to eventually lead to a universally-quantified inductive invariant, if one exists.

CHAPTER 3

Word-level Verification by Equality Abstraction of Data State

While bit-level IC3-based algorithms for hardware model checking represent a major advance over prior approaches, their reliance on propositional clause learning using SAT solvers poses scalability issues for RTL designs with wide datapaths and complex word-level operations. In this chapter, we present a novel technique that combines incremental induction with *equality abstraction* (EA) to allow scalable word-level model checking using SMT solvers. EA defines the abstraction implicitly using equality relations among word-level terms present in the RTL description of the input problem. Additionally, EA aligns perfectly with EUF abstraction where wide operations are abstracted with uninterpreted functions using EUF logic to allow abstract reasoning completely independent of the design's bit widths. We show how to efficiently implement word-level IC3 with EA (WIC3+EA), both with interpreted and uninterpreted functions, to offer scalable model checking of control-centric problems irrespective of the problem size or complexity of operations.

The chapter is organized as follows: §3.1 presents a brief motivation of the approach, with §3.2 explaining equality abstraction in detail. §3.4 shows how EA is efficiently integrated within the word-level IC3 framework. §3.5 describes the WIC3+EA algorithm, and the correctness of the complete methodology is proved in §3.6. §3.7 provides a preliminary evaluation of WIC3+EA on a suite of open-source and industrial Verilog RTL designs, with a more detailed evaluation and a collection of complementary techniques presented in the next chapter.

3.1 Motivation

Consider a predicate $p := (a + b < 1)$ defined over two 32-bit variables a and b. An equivalent propositional-level representation of p will involve a bit-blasted expression involving 64 Boolean variables and several hundred propositional clauses. Consequently, bit-level model checking algorithms struggle with the *state-space explosion* [89, 86, 83] as the variable bit widths increases.

The word-level representation of a problem contains very useful, high-level information that can be exploited to offer better scalability. Specifically, an *implicit* equality abstraction based on equality reasoning can be automatically inferred using terms built from objects present in the word-level description (like a, b, 1, $+$, $<$), which ignores the exact bit-precise data state while still retains the most useful relations on word-level terms, e.g., $a \neq b$, $b = 1$, $a + b \neq 1$. The approach can be further combined with EUF abstraction [63, 38] to simplify reasoning for the underlying query solver. This, coupled with efficient SMT solving, allows for an effective word-level model checking algorithm that can scale better than bit-level engines for a variety of verification problems. Moreover, the word-level induction-based procedure has the unique strength of producing word-level proof certificates that enable provable assurance. These word-level inductive proofs are useful in a variety of applications [118, 139], including techniques to verify distributed protocols automatically, that we discuss later in Chapter 5.

3.2 Equality Abstraction

Recall predicate abstraction (PA) [132], that encodes the abstract state space using a set of predicates whose Boolean assignments encode the abstract states. In contrast, equality abstraction (EA) encodes the abstract state space using equality relations on the set of terms present in the word-level structure of the problem.[1] Abstract states in EA correspond to partition assignments that capture *all equality relations* among the problem's terms, thus making equality reasoning the core of the abstraction. The relevant parts of the abstract transition relation in both *implicit* PA [79, 80] and EA are constructed incrementally, as needed, during the reachability search using solver queries. We will use $\mathcal{P}$ to denote the original *concrete* problem and $\hat{\mathcal{P}}$ to denote its version in the EA domain. Models in $\mathcal{P}$ use the interpretation $\mathcal{I}_B$, i.e., exact bitvector assignments, whereas models in $\hat{\mathcal{P}}$ use the $\mathcal{I}_P$ interpretation, i.e., partition assignments. Effectively, EA hides away irrelevant bit-precise details and can infer higher-level equality relations among the terms in the problem description. The abstract problem $\hat{\mathcal{P}}$ is able to capture the important relations that are structurally most relevant to the problem, i.e., equality and disequality among terms in the word-level description, instead of worrying about exact bit-precise assignments.

[1] Note that in [124, 125], we referred this abstraction as syntax-guided abstraction. We relabelled it as equality abstraction to emphasize on the significance of equality relations central to this technique.

Example 1: Consider the hardware design shown in Figure 1.1. The model checking problem $\mathcal{P}$ corresponding to this design can be extracted as:

$$\begin{aligned}
&\mathcal{P} = \langle X, \mathit{Init}, T, P\rangle \\
&X = \{u, v\} \\
&\mathit{Init} = (u = 1) \wedge (v = 1) \\
&T = (u' = \mathit{ite}(u < v, u + v, v + 1)) \wedge (v' = v + 1) \\
&P = ((u + v) \neq 1)
\end{aligned}$$

Assume u and v to be k-bit wide. $\mathcal{P}$ has 1 predicate $u < v$ and 5 words $\{1, u, v, u + v, v + 1\}$. Consider a concrete state $s := (u, v) = (1, 2)$. Its corresponding abstract state is obtained by evaluating the problem's terms using the concrete state assignment and creating a partition assignment based on these evaluations. In this example, the abstract state is easily seen to be:[2]

$$\hat{s} := \{u < v, \top \mid \perp\}_1,\ \{1, u \mid v \mid u + v, v + 1\}_k \tag{3.1}$$

The abstract state $\hat{s}$ consists of two partitions, one for Boolean terms $u < v$, $\top$ and $\perp$, and the other for k-bit wide terms 1, u, v, $u + v$ and $v + 1$, where "|" separates the different equivalence classes in a partition. Boolean constants, i.e., $\top$ and $\perp$ are implicitly present in $\mathcal{P}$ and are always included in all partition assignments. Note that the partition of Boolean terms will always consist of exactly two equivalence classes, one each corresponding to $\top$ and $\perp$.

The biggest advantage of EA is that the abstract state-space size is completely independent of the bit width of variables and is only dependent on the number of terms in the word-level description. Given $\mathcal{P}$ with, say, m total state bits, the concrete system has 2^m states. On the other hand, the total number of abstract states is bounded by $2^p \times B_n$, where p is the number of predicates in $\hat{\mathcal{P}}$, n is the number of words in $\hat{\mathcal{P}}$, and B_n is the n^{th} Bell number [218] (the number of unique partitions on n terms). For example, let $k = 16$ in Example 1. The size of the concrete state space is $2^{2\times 16}$ i.e., ~ 4.2 billion, while the number of abstract states is $2^1 \times B_5 = 104$, completely independent of k.

Given a satisfiable formula φ, a concrete theory solver (i.e., QF_BV solver) produces a bitvector solution s that assigns each term in the problem with an exact bit-precise value. We can perform equality abstraction using a post-processing *partition building* step, where the abstract solution $\hat{s}$ in the EA domain is expressed as a partition assignment on terms in φ. The partition builder returns the abstract solution $\hat{s} \models_{abstract} \varphi$ *iff* there exists a bitvector assignment s such that $s \models_{concrete} \varphi$

[2] In this notation, vertical bars separate the equivalence classes of the partition. Thus $\{a, b|c\}_N$ should be interpreted to mean $\{\{a, b\}, \{c\}\}$ in the standard notation for partitions, where a, b, and c are N-bit wide. A partition assignment is therefore denoted by a list of partitions, one each corresponding to each bit width in the problem.

and $\hat{s} = \alpha(s, \varphi)$, where α is the abstraction function that converts a bitvector assignment s to a partition assignment on terms in φ. In practice, the partition building step can be easily implemented using a simple evaluation of each term in the formula to construct a partition assignment based on the bitvector assignment.

Example 2: Consider $\mathcal{P}$ from Example 1. Let $k = 2$. Consider the formula $\varphi = P \wedge T \wedge \neg P'$ and a satisfying concrete solution $s := (u, v, u', v') = (0, 2, 2, 3)$. Terms in φ evaluate as $(u < v, u + v, v + 1, u' + v') = (\top, 2, 3, 1)$ under s, resulting in the abstract solution to be:

$$\hat{s} := \{u < v, \top \mid \perp\}_1,\ \{u \,|\, 1, u' + v' \,|\, v, u + v, u' \,|\, v + 1, v'\}_2$$

We can always construct a *unique* abstract solution $\hat{s}$ given a formula φ and its concrete solution s. Modern SMT solvers (e.g. [107, 97]) have support to give the bitvector assignment for each term in the formula without any extra cost. Terms with the same assigned value go in the same equivalence class of a partition, while different assignments mean different classes. An abstract solution is *complete* if it contains all terms present in $\mathcal{P}$. The equality-abstracted state space is defined by the universe of complete abstract solutions.

To better understand how EA compares to PA, consider $\mathcal{P}$ from Example 2. There are 16 concrete states in $\mathcal{P}$. The four predicates in $\mathcal{P}$, i.e., p_1: $(u = 1)$, p_2: $(v = 1)$, p_3: $((u + v) = 1)$ and p_4: $(u < v)$ are a natural choice as initial predicates for PA. Table 3.1 compares the abstract domain for EA and PA against the concrete state space. PA with p_{1-4} as predicates partitions the concrete states into 9 abstract states. EA on the other hand offers higher expressiveness and partitions the concrete states into 13 abstract states.

An abstract solution can be *projected* on any subset of symbols, call it a *projection set*, by *co-factoring* the solution to eliminate all terms with any symbol outside the projection set, i.e., by simply dropping terms from the partition assignment that contain a symbol outside the projection set. An abstract solution can also be easily converted to an equivalent cube by adding all equality and disequality relations needed to *cover* the solution.

Example 3: Consider $\hat{s}$ from Example 2. $\hat{s}$ can be projected on the projection set $\sigma = \{+, 1, u', v'\}$ to get a *partial* abstract solution representing the destination states as $\hat{s}|_\sigma := \{1, u' + v' \,|\, u' \,|\, v'\}_2$. The corresponding cube representation of $\hat{s}|_\sigma$ is:

$$\mathit{cube}(\hat{s}|_\sigma) = ((u' + v') = 1) \wedge (u' \neq 1) \wedge (u' \neq u' + v') \wedge (v' \neq 1) \wedge (v' \neq u' + v') \wedge (u' \neq v')$$

With EA, the abstract state space induces a partition on the concrete state space such that each concrete state is mapped to a single abstract state. An abstract state, thus, corresponds to a set of

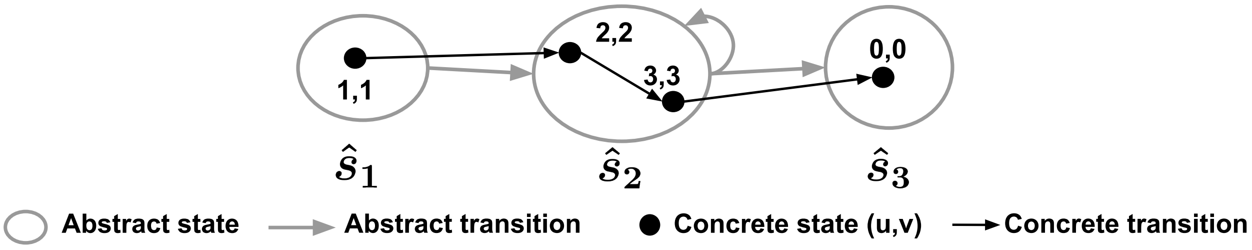
Table 3.1: Mapping of abstract states on concrete states for EA and PA

Index	**EA**: partition assignment on	**Concrete**	**PA**:
	$u < v, 1, u, v, u+v, v+1$	**states**: (u, v)	$p_1 p_2 p_3 p_4$
1	$\{\top \mid u < v, \bot\}_1,\ \{1, v+1 \mid u, v, u+v\}_2$	$(0,0)$	0000
2	$\{\top \mid u < v, \bot\}_1,\ \{1, v+1 \mid u, u+v \mid v\}_2$	$(2,0)$, $(3,0)$	
3	$\{\top \mid u < v, \bot\}_1,\ \{1 \mid u, v \mid u+v \mid v+1\}_2$	$(2,2)$, $(3,3)$	
4	$\{u < v, \top \mid \ \bot\}_1,\ \{1 \mid u \mid v, u+v \mid v+1\}_2$	$(0,2)$	0001
5	$\{u < v, \top \mid \ \bot\}_1,\ \{1 \mid u, v+1 \mid v, u+v\}_2$	$(0,3)$	
6	$\{\top \mid u < v, \bot\}_1,\ \{1, u+v \mid u, v+1 \mid v\}_2$	$(3,2)$	0010
7	$\{u < v, \top \mid \ \bot\}_1,\ \{1, u+v \mid u \mid v \mid v+1\}_2$	$(2,3)$	0011
8	$\{\top \mid u < v, \bot\}_1,\ \{1, v \mid u, v+1 \mid u+v\}_2$	$(2,1)$	0100
9	$\{\top \mid u < v, \bot\}_1,\ \{1, v \mid u \mid u+v \mid v+1\}_2$	$(3,1)$	
10	$\{u < v, \top \mid \ \bot\}_1,\ \{1, v, u+v \mid u \mid v+1\}_2$	$(0,1)$	0111
11	$\{u < v, \top \mid \ \bot\}_1,\ \{1, u \mid v \mid u+v, v+1\}_2$	$(1,2)$, $(1,3)$	1001
12	$\{\top \mid u < v, \bot\}_1,\ \{1, u, u+v, v+1 \mid v\}_2$	$(1,0)$	1010
13	$\{\top \mid u < v, \bot\}_1,\ \{1, u, v \mid u+v, v+1\}_2$	$(1,1)$	1100
	others: ($2^1 \times B_5 - 13 = 91$)	infeasible	others: ($2^4 - 9 = 7$)

concrete states causing the abstract transition relation to be *non-deterministic*. Therefore, EA is *sound* but may lead to *spurious* behavior, leading to an abstract path to be discontinuous and not have a corresponding feasible concrete path.

Example 4: Consider the following abstract path $\hat{\pi} = \langle \hat{s}_1, \hat{s}_2, \hat{s}_3 \rangle$ from $\mathcal{P}$ in Example 2:

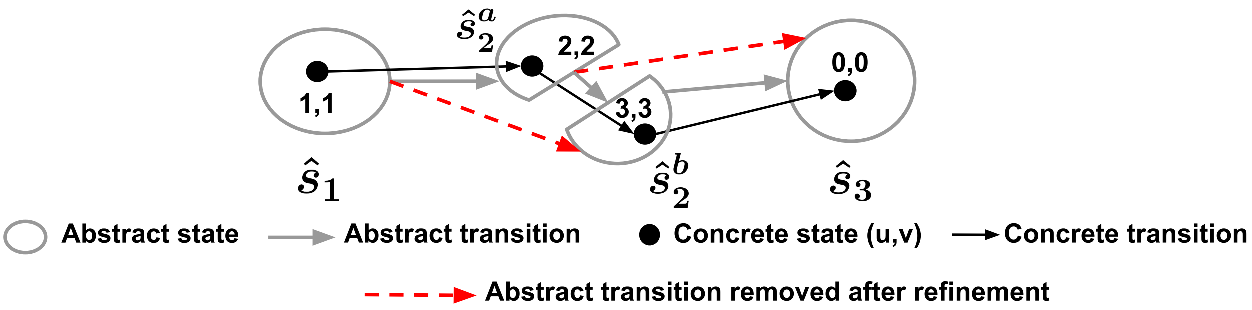
$$\hat{s}_1 := \{\top \mid u < v, \bot\}_1,\ \{1, u, v \mid u+v, v+1\}_2$$
$$\hat{s}_2 := \{\top \mid u < v, \bot\}_1,\ \{1 \mid u, v \mid u+v \mid v+1\}_2$$
$$\hat{s}_3 := \{\top \mid u < v, \bot\}_1,\ \{1, v+1 \mid u, v, u+v\}_2$$



$\hat{s}_1$ has a concrete transition to $\hat{s}_2$, $\hat{s}_2$ can concretely transition to $\hat{s}_3$, though there isn't a continuous 2-step concrete path from $\hat{s}_1$ to $\hat{s}_3$ via $\hat{s}_2$.

EA can be refined by adding new terms to the abstract problem $\hat{\mathcal{P}}$. For example, adding the symbolic term $u + v + 1$ eliminates the spurious behavior of Example 4 by breaking $\hat{s}_2$ into two distinct refined states $\hat{s}_2^a$ and $\hat{s}_2^b$:

$$\hat{s}_2^a := \{\top \,|\, u < v, \bot\}_1,\ \{1, u+v+1 \,|\, u, v \,|\, u+v \,|\, v+1\}_2$$
$$\hat{s}_2^b := \{\top \,|\, u < v, \bot\}_1,\ \{1 \,|\, u, v, u+v+1 \,|\, u+v \,|\, v+1\}_2$$



The addition of the new term $u + v + 1$ results in adding more granularity to the abstract state space by adding, in a *single* refinement step, all equality relations between the new term $u+v+1$ and already-present terms 1, u, v, $u + v$ and $v + 1$. This eliminates the spurious 2-step abstract path $\hat{\pi}$ in the new abstract domain after refinement. Similarly, adding the constant term 2 (or 3) instead of the symbolic term $u+v+1$ has the same effect of eliminating the spurious abstract path $\hat{\pi}$.

Equality abstraction has the following advantages over PA:

- Unlike predicate abstraction or its variants [132, 41], EA is *implicitly* defined by the original word-level structure and does not require a user-specified set of initial predicates or solver queries to generate the abstract state space.
- By construction, EA accounts for all equality and disequality relations among terms in the structure and offers higher granularity and expressiveness than implicit predicate abstraction [79, 80], resulting in less spurious behavior.
- EA is refined by adding new terms that are absent in the original problem syntax, while PA relies on adding new predicates for refinement. Furthermore, equality reasoning, which is at the heart of EA, allows all equality relations involving a newly-introduced term to be automatically detected; in PA such relations are discovered one by one in multiple refinement iterations.

3.3 Equality Abstraction with Uninterpreted Functions

EA with bit-precise QF_BV queries can be dubbed as equality with *interpreted* functions, i.e., EIF abstraction. EIF may, however, not scale well to large problems with complex operations. Since equality abstraction only requires a partition assignment on terms and not exact bitvector assignments, it aligns perfectly with equality with *uninterpreted* functions [63, 38, 34, 177], i.e., EUF abstraction, where all data operations (like arithmetic, bitwise, shift, etc.) are further abstracted to uninterpreted functions. EUF abstraction is most appropriate for control-centric properties where correctness is largely independent of data state. Word-level incremental induction with EUF abstraction allows for efficient reasoning using QF_UF queries regardless of the bit width of variables or complexity of data operations.

Example 5: Consider $\mathcal{P}$ from Example 1. Using EUF abstraction, the abstract problem becomes:

$$\bar{\mathcal{P}} = \langle \bar{X}, \bar{Init}, \bar{T}, \bar{P} \rangle$$
$$\bar{X} = \{\bar{u}, \bar{v}\}$$
$$\bar{Init} = (\bar{u} = \bar{1}) \wedge (\bar{v} = \bar{1})$$
$$\bar{T} = (\bar{u}' = ite(LT(\bar{u}, \bar{v}), ADD(\bar{u}, \bar{v}), ADD(\bar{v}, \bar{1})) \wedge (\bar{v}' = ADD(\bar{v}, \bar{1}))$$
$$\bar{P} = (ADD(\bar{u}, \bar{v}) \neq \bar{1})$$

Here, the abstract problem $\bar{\mathcal{P}}$ uses uninterpreted sorts instead of bitvectors (indicated by $^{-}$). All data operations (i.e., $<$, $+$) are abstracted to uninterpreted functions (i.e., LT, ADD), while ground terms are converted to uninterpreted functions of 0-arity. Since equality abstraction still limits UF instances to terms derived from the original problem, it does not face any non-termination issue due to unbounded function compositions.

3.4 Word-level IC3 with Equality Abstraction

Similar to [77, 144, 79, 177, 80], word-level IC3 with equality abstraction, or WIC3+EA in short, uses SMT solving to raise reasoning from propositional to first-order logic, by combining the incremental induction algorithm with a counterexample-guided abstraction refinement (CEGAR) loop [159, 81]. The core incremental induction algorithm is performed in the EA domain. There are 3 key differences between WIC3+EA and the well-studied bit-level IC3 [58, 108].

- How to **integrate equality abstraction** with incremental induction?
- How to transform a single counterexample-to-induction to a **generalized cube**?

- How to **refine** spurious counterexamples?

We elaborate on these key differences in this section. All other concepts in incremental induction, including forward propagating clauses from a frame to the next, expanding an unreachable cube using unsat-core from the solver through *minimal unsatisfiable subsets* (MUS) extraction [181], etc., remain identical to the bit-level procedure and can be equivalently applied in WIC3+EA using word-level clauses and SMT solvers.

3.4.1 Integrating Equality Abstraction

At the word level, incremental induction using equality abstraction with interpreted functions (i.e., EIF abstraction) is performed using satisfiability queries formulated in the first-order logic (FOL) and solved using a SMT solver with `QF_BV` theory support. The solution returned by the SMT solver for a satisfiable query is composed of bit-precise assignments to variables in the design. Parition builder operates on these bit-exact values and transforms the bitvector assignments to an abstract solution over word-level terms in the design. An abstract solution distributes the word-level terms into equivalence partitions, one for each bit width in the design, through an evaluation of terms in the query to create a partition distribution. Example 2 from Section 3.2 provides an example of this step. Alternatively, WIC3+EA using equality with uninterpreted functions (i.e., EUF abstraction) employs a SMT solver with `QF_UF` theory support, which already returns word-level terms partitioned into equivalence classes as the abstract solution.

3.4.2 Structure-guided Cube Generalization

Consider the standard 1-step backward reachability query $SAT\ ?\ [F_m \wedge T \wedge c']$ during incremental induction. This query is satisfiable *iff* there exists a transition from the current overapproximation of reachable abstract states at depth m (i.e., frame F_m) to a given destination cube c'. If the query is satisfiable, the solution to the query provides single evidence of a CTI, i.e., an abstract state that transitions to the destination cube c'. As explored in previous works [108, 58, 56], it is essential for the performance of the incremental induction algorithm to *generalize* the single CTI derived from the particular solver solution into a generalized cube c_{gen}, that compactly encodes multiple counterexamples to induction. For the propositional case using bit-level IC3, the authors of [108] suggest *ternary simulation* to generalize the particular solution into a cube. This generalization, as well as cube generalization suggested in the original IC3 algorithm [58], ensures *strict continuity*.

Definition 1. *(Strict Continuity) Given a destination cube c', every state in the generalized cube c_{gen} should have a transition under T to c', i.e., $\forall_s\ SAT\ ?\ [s \wedge T \wedge c']$ is satisfiable, where $\{s \in c_{gen} \mid s \text{ is a state } \}$.*

Strict continuity is not necessary for the incremental induction algorithm, though it is sufficient to guarantee "relaxed" *continuity*.

Definition 2. *(Continuity) Given a sequence of cubes $\mathcal{C} = \langle c_m, \ldots, c_n \rangle$ with $c_n = \neg P$, there exists a path $\pi = \langle s_m, \ldots, s_n \rangle$ such that $\{s_i \in c_i \mid s_i$ is a state $\}$ for all $i \in \{m, \ldots, n\}$.*

For correctness, the necessary condition for any cube generalization procedure is to ensure *continuity* (Def. 2), i.e., there should exist a path from the generalized cube c_{gen} to $\neg P$. After all, any cube with a continuous path to a bad state (i.e., a state satisfying $\neg P$) needs to be checked for reachability from the initial states.

It is unclear how to extend ternary simulation to word-level semantics since ternary simulation inherently relies on modeling the system as a Boolean circuit. Instead, we propose a *structure-guided* cube generalization technique that we call StructGen, which exploits the word-level structure of the problem to *cheaply* generalize a single abstract solution $\hat{s}$ *without* any additional solver reasoning. The procedure exploits structural *cone-of-influence* (COI) and model-based justification step (JustifyCOI) to identify *relevant* portions that are sufficient to *justify* the single abstract solution, analogous to justification in test pattern generation [225]. JustifyCOI creates a projection set σ consisting of only relevant symbols that are structurally needed to reach the destination cube c'. The abstract solution $\hat{s}$ is projected on these relevant symbols to get the generalized cube c_{gen} as $cube(\hat{s}|_\sigma)$.

Algorithm 1 presents StructGen, our proposed structure-guided cube generalization procedure employing COI with model-based justification. Given a particular abstract solution $\hat{s}$ and the destination cube c', the procedure traverses the *concrete* structural COI of c' and collects symbols encountered in the process (line 3, 7-25). The key idea is that during the traversal we can syntactically prune away portions that are not important under the given particular solution (lines 12, 16) and only visit portions that justify leading to the destination. Once the relevant symbols are collected, the algorithm projects $\hat{s}$ on these symbols to get the generalized cube (lines 5-6).

Algorithm 1 guarantees abstract continuity (Def. 2), with the generalized cube always having an *abstract* path to $\neg P$ in $\hat{\mathcal{P}}$. The algorithm however does not guarantee strict continuity (Def. 1), as evident from the following example:

Algorithm 1 Structure-guided Cube Generalization

```
procedure StructGen($\hat{s}$, $c'$)    ▷ $\hat{s}$ is a particular abstract solution, $c'$ is a destination cube
    $\sigma \leftarrow \sigma_{refine}$    ▷ initialize projection set (initially $\sigma_{refine} = \emptyset$)
    JustifyCOI($\hat{s}$, $c'$, $\sigma$)    ▷ build projection set $\sigma$
    $\sigma \leftarrow \sigma - X'$    ▷ get rid of next state symbols
    $\hat{s}|_\sigma \leftarrow$ Project($\hat{s}$, $\sigma$)    ▷ project $\hat{s}$ on $\sigma$
    return $cube(\hat{s}|_\sigma)$    ▷ convert to a cube and return

procedure JustifyCOI($\hat{s}$, $\varphi$, $\sigma$)    ▷ $\varphi$ is a FOL expression, $\sigma$ is passed by reference
    if $\varphi$ is a conditional operation then    ▷ if $\varphi$ is an if-then-else expression
        $\langle cond, arg_\top, arg_\bot \rangle \leftarrow$ BreakCondition($\varphi$)    ▷ get condition and arguments
        JustifyCOI($\hat{s}$, $cond$, $\sigma$)    ▷ repeat on condition $cond$
        $val \leftarrow$ Evaluate($cond$, $\hat{s}$)    ▷ evaluate $cond$ under $\hat{s}$
        JustifyCOI($\hat{s}$, $(val = \top)$ ? $arg_\top$ : $arg_\bot$, $\sigma$)    ▷ repeat only on the relevant branch
    else if $\varphi$ is a logical operation then
        $val \leftarrow$ Evaluate($\varphi$, $\hat{s}$)    ▷ evaluate $\varphi$ under $\hat{s}$
        if IsControlling($val$, $\varphi$) then    ▷ if assigned a controlling value ($\bot$ for $\wedge$, $\top$ for $\vee$)
            JustifyCOI($\hat{s}$, GetControlling($\varphi$, $\hat{s}$), $\sigma$)    ▷ repeat only on controlling arg.
        else
            for each $a \in$ Argument($\varphi$) do
                JustifyCOI($\hat{s}$, $a$, $\sigma$)    ▷ repeat on all arguments
    else
        for each $a \in$ Argument($\varphi$) do
            JustifyCOI($\hat{s}$, $a$, $\sigma$)    ▷ repeat on all arguments
        if $\varphi$ is a next state variable then
            JustifyCOI($\hat{s}$, GetRelation($\varphi$), $\sigma$)    ▷ get the next state relation for $\varphi$ from $T$
        Add symbol($\varphi$) to $\sigma$    ▷ add symbol of $\varphi$ to the projection set
```

Example 6: Let $\mathcal{P}$ be a model checking problem such that:

$$\mathcal{P} = \langle X, Init, T, P \rangle$$

$$X = \{u, v, w\}$$

$$Init = (u = 1) \wedge (v = 1) \wedge (w = 1)$$

$$T = (u' = ite((u < v) \vee (v < w), u + v, v + 1)) \wedge (v' = v + 1) \wedge (w' = w + 1)$$

$$P = ((u + v) \neq 1)$$

where u, v, w are 3-bit wide. Consider the following query and its particular solution:

$$F_1 = P$$

$$\varphi = F_1 \wedge T \wedge \neg P'$$

$$Q_1 := SAT\ ?\ [\varphi]\text{, which returns satisfiable with bit-precise solution } s$$

$$s = (u, v, w, u', v', w') = (0, 4, 2, 4, 5, 3) \qquad\qquad \hat{s} = \alpha(\varphi, s)$$

$$\hat{s} = \{u < v, \top \,|\, v < w, \bot\}_1,\ \{u \,|\, 1, u' + v' \,|\, w \,|\, w + 1, w' \,|\, v, u + v, u' \,|\, v + 1, v'\}_3$$

StructGen($\hat{s}$, $\neg P'$) creates the generalized cube c_{gen} as follows:

$$\begin{aligned}
\sigma &= \{+, u', v', 1, <, u, v\} - \{u', v', w'\} \\
&= \{+, <, 1, u, v\} \\
\hat{s}|_\sigma &= \{u < v, \top \,|\ \bot\}_1,\ \{u \,|\, 1 \,|\, v, u + v \,|\, v + 1\}_3 \\
c_{gen} &= cube(\hat{s}|_\sigma) \\
&= (u < v) \wedge (v = u + v) \wedge (u \neq 1) \wedge (u \neq v) \wedge (u \neq u + v) \wedge (u \neq v + 1) \\
&\quad \wedge (1 \neq v) \wedge (1 \neq u + v) \wedge (1 \neq v + 1) \wedge (v \neq v + 1) \wedge (u + v \neq v + 1)
\end{aligned} \tag{3.2}$$

On careful analysis one can see that not all abstract states in c_{gen} have an abstract transition to the destination ($\neg P'$). For example, consider the abstract state:

$$\hat{a} = \{u < v, \top \,|\, v < w, \bot\}_1,\ \{u \,|\, 1, w \,|\, v, u + v, w + 1 \,|\, v + 1\}_3$$

$\hat{a}$ is an abstract state in the cube c_{gen}, but it does not have a transition under T to any destination state, i.e., $SAT\ ?\ [cube(\hat{a}) \wedge T \wedge \neg P']$ is unsatisfiable.

Non-determinism in the EA domain is the reason why Alg. 1 does not follow strict continuity. Even though Def. 1 is violated, Alg. 1 still guarantees "relaxed" continuity (i.e., Def. 2) in the abstract domain. This is because StructGen ensures that all terms in $\hat{\mathcal{P}}$ that are required to lead to the destination c' under the abstract solution $\hat{s}$ are retained in the generalized cube c_{gen} as is from $\hat{s}$. As a result, even though c_{gen} has abstract states that do not have an abstract transition to the destination c', this cannot result in an abstract path discontinuity while still limiting to terms in $\hat{\mathcal{P}}$. JustifyCOI acts as a quick sweeper that removes irrelevant terms that will never get involved with any query that satisfies $c_{gen} \wedge T \wedge c'$, and encodes the sufficient information by projecting the abstract solution $\hat{s}$ on the relevant symbols as $\hat{s}|_\sigma$.

Our proposed structure-guided cube generalization procedure has the following advantages:

- In contrast to solver-based methods suggested in [76, 144, 56], StructGen using COI-based justification is inexpensive since it does not require any solver query.

- Since cube generalization is driven from the syntactic cone of the destination, the procedure only captures the relevant information leading to a bad state.
- The technique guarantees continuity with no need for *lifting refinement* [56].
- Unlike [177, 176], the technique does not use weakest preconditions (WP) for generalization, which can be regarded as implicitly unrolling the transition relation. WP-based techniques generate new terms through function compositions, which complicates the abstract state space and can often cascade to cause incompleteness and poor SMT solving.

Structure-based cube generalization offers an inexpensive and effective procedure to expand a single solver solution to a set of solutions for word-level incremental induction. An identical generalization procedure can be used for EA with uninterpreted functions, and possibly, even for predicate abstraction.

3.4.3 Refining Equality Abstraction

Running incremental induction in the EA domain either generates an inductive invariant that proves the property to be true or produces an abstract counterexample evidence $\mathcal{C}$. An abstract counterexample $\mathcal{C}$ of length $n-1$ is represented by a sequence of n abstract cubes $\langle c_1, c_2, \ldots, c_n \rangle$, where $c_n = \neg P$.

As shown earlier through Example 4 in Section 3.2, $\mathcal{C}$ can be spurious when the terms in the original problem are insufficient to express the bit-precise nature of the concrete problem, resulting in a *concrete path discontinuity*. One way to identify spurious behavior in $\mathcal{C}$ is by checking the satisfiability of a single concrete path query along $\mathcal{C}$ with explicit unrolling, i.e., $SAT\ ?\ [Init \wedge \big(\bigwedge_{i=1}^{n-1} c_i^i \wedge T^i\big) \wedge c_n^n]$ (where φ^i denotes the formula φ at i^{th} transition step) using $\texttt{QF\_BV}$ SMT solving. Checking satisfiability of such a query with multiple copies of T is not scalable in practice as the length of $\mathcal{C}$ increases. We instead perform *incremental* refinement *along the counterexample* which uses 1-step queries to perform *forward image computation* [134] along $\mathcal{C}$.

Algorithm 2 presents the incremental refinement procedure to refine EA by detecting any spurious behavior in the abstract counterexample $\mathcal{C}$. The procedure formulates at most $n-1$ queries $Q_i := SAT\ ?\ [p_{i-1} \wedge c_{i-1} \wedge T \wedge c_i']$ for $i = 2 \ldots n$ such that $p_1 = Init$, and p_i equals the *symbolic post image* [134, 195] of $p_{i-1} \wedge c_{i-1}$ under the bit-precise solution of the query Q_i (lines 2-6). To compute p_i after a satisfiable query Q_i, say with bitvector solution s, fresh symbolic constants are used to replace unconstrained variables at that step and *syntactically* evaluate T under s to get the symbolic post image of $p_{i-1} \wedge c_{i-1}$ for the next step (line 6). This generates new terms and results in an *implicit* unrolling of T along $\mathcal{C}$. The concrete, i.e., $\texttt{QF\_BV}$ solver, checks the satisfiability of Q_i in increasing order (from $i = 2$ to n) and stop as soon as a query is found unsatisfiable (lines

Algorithm 2 Refinement of Equality Abstraction

```
1  procedure RefineEA($\mathcal{C}$)
2      $p_1 \leftarrow Init$
3      for $i = 2$ to $n$ do
4          $\psi_i \leftarrow p_{i-1} \wedge c_{i-1} \wedge T \wedge c'_i$
5          if SAT ? $[\psi_i]$: solution $s$ then
6              $p_i \leftarrow$ PostImage($p_{i-1} \wedge c_{i-1}, s$)    ▷ compute image($p_{i-1} \wedge c_{i-1}$) under $s$
7          else                                              ▷ i.e., $\mathcal{C}$ is spurious
8              $m \leftarrow$ MUS($\psi_i$)                  ▷ find MUS for the UNSAT query
9              $m \leftarrow$ Substitute($m$)                ▷ eliminate symbolic constants
10             $\Phi \leftarrow \neg m$
11             $T \leftarrow T \wedge \Phi$                  ▷ conjoin lemma to $\hat{T}$
12             $\sigma_{new} \leftarrow$ symbols(NewTerms($\Phi$))    ▷ find symbols in new terms
13             $\sigma_{refine} \leftarrow \sigma_{refine} \cup \sigma_{new}$    ▷ add permanent symbols
14             return $\emptyset$
15     return $\mathcal{C}$                                  ▷ i.e., $\mathcal{C}$ is a true counterexample
```

3-14). From the unsatisfiable query, RefineEA extracts a minimal unsatisfiable subset [208, 181] (MUS) m and get rid of any symbolic constant in m using substitution, or alternatively, using instantiation with the last solver assigned value if substitution is not possible (lines 8-9). Since the unsatisfiability is due to a *concrete path infeasiblity*, m necessarily contains constraints from the forward image computation that include new terms generated from substitution/instantiation. These new terms can be exact constants (e.g., 2) or symbolic terms (e.g., $u + (v + 1)$), and are important to eliminate the spurious counterexample $\mathcal{C}$. RefineEA adds these newly discovered terms to the EA domain by deriving a *refinement (path) lemma* by negating m (line 10). These new terms are introduced to the abstract problem $\hat{\mathcal{P}}$ in the EA domain by conjoining the refinement lemma to the transition relation T (line 11).

New terms created are crucial to eliminate spurious counterexamples. They were absent in the original problem and hence the EA domain wasn't expressive enough to capture infeasibilities involving them. Adding the refinement lemma with these new terms automatically augments the abstract problem and makes them part of future iterations of wIC3+EA. Symbols corresponding to the new terms are added as permanent members of all projection sets computed using Alg. 1 to ensure that future iterations of StructGen doesn't ambitiously generalize them away (lines 12-13). This is essential since these new terms are not part of the original problem structure but are required to eliminate spurious counterexamples.

If all queries Q_i are satisfiable, it means that $\mathcal{C}$ is indeed including a true counterexample that disprove the property. One instance of a true counterexample can be easily retrieved by keeping track of solutions to the queries Q_i.

After learning a refinement lemma, wIC3+EA *incrementally* resumes the word-level incremental induction procedure from the last top frame. Since the abstraction-refinement procedure is completely *monotonic* with each iteration making the abstract domain more precise and finer by adding new terms, we can reuse all reachability information and abstract clauses from previous iterations.

The refinement procedure provides the following advantages:

- All concrete queries involve a single instance of the transition relation and avoids explicit unrolling.
- There is no path explosion since the refinement is constrained to only the path(s) along the abstract counterexample.
- Symbolic constants for unconstrained variables allow avoiding enumerative simulation on exact variable assignments returned by the solver.
- The procedure is completely incremental and allows reuse of all previous abstract clause learning.

3.4.4 Refining EA with Uninterpreted Functions

Equality abstraction with uninterpreted functions can introduce additional spurious behavior with inconsistencies resulting from the usage of uninterpreted operations instead of interpreted concrete operations. Given $\bar{\mathcal{C}} = \langle \bar{c}_1, \ldots, \bar{c}_n \rangle$, we can check for such inconsistencies using at most $n-1$ concrete bitvector queries $Q_i^d := SAT\ ?\ [c_{i-1} \wedge T \wedge c_i']$ for $i = 2 \ldots n$ in any order, similar to [177]. In the case any query returns UNSAT, a *refinement (data) lemma* is learned to constrain the abstract transition relation $\bar{T}$. Since data lemmas are generated from 1-step queries, they never add any new term and therefore will never increase the size of the abstract state space. They eliminate spurious abstract states/transitions that got introduced due to data abstraction with uninterpreted functions, while path lemmas, on the other hand, add more granularity by introducing new terms.

Example 7: Consider $\bar{\mathcal{P}}$ from Example 1. wIC3+EA with EUF abstraction produces an abstract counterexample $\bar{\mathcal{C}} = \langle \bar{c}_1, \bar{c}_2, \bar{c}_3 \rangle$ as follows:

$$\begin{aligned}
\bar{c}_1 &:= \neg LT(\bar{u}, \bar{v}) \wedge \{\bar{1}, \bar{u}, \bar{v} \mid ADD(\bar{v}, \bar{1}), ADD(\bar{u}, \bar{v})\} \\
\bar{c}_2 &:= \neg LT(\bar{u}, \bar{v}) \wedge \{\bar{1} \mid \bar{u}, \bar{v} \mid ADD(\bar{v}, \bar{1}) \mid ADD(\bar{u}, \bar{v})\} \\
\bar{c}_3 &:= \{\bar{1}, ADD(\bar{u}, \bar{v}) \mid \bar{u}, \bar{v}\}
\end{aligned}$$

We can eliminate spurious counterexample $\bar{\mathcal{C}}$ as follows:

$$
\begin{aligned}
&Q_3^d := SAT\ ?\ [\ c_2 \wedge T \wedge c_3'\ ] \text{ gives UNSAT with MUS } m\\
&m = \neg(u < v) \wedge T \wedge ((u' + v') = 1)\\
&\bar{m} = \neg LT(\bar{u}, \bar{v}) \wedge \bar{T} \wedge (ADD(\bar{u}', \bar{v}') = \bar{1})\\
&\bar{\Phi} = \neg \bar{m}\\
&\bar{T} = \bar{T} \wedge \bar{\Phi}
\end{aligned}
$$

In the case all concrete queries Q_i^d are satisfiable, it means that there is no inconsistency due to uninterpreted functions, though concrete path discontinuity due to equality abstraction still needs to be checked using Alg. 2.

3.5 wIC3+EA Algorithm

Algorithm 3 presents the detailed pseudo code of word-level IC3 using equality abstraction with interpreted functions. Given a model checking problem $\mathcal{P} = \langle X, \mathit{Init}, T, P\rangle$, wIC3+EA wraps the incremental induction algorithm (line 5) in a CEGAR framework (lines 4-11). Upon termination, wIC3+EA either a) produces an inductive invariant Inv that proves the property for $\mathcal{P}$, or b) a true counterexample trace $\mathcal{C}$ that serves as a witness to its violation in $\mathcal{P}$ (lines 4-11).

wIC3 presents the word-level IC3 procedure (lines 12-54). wIC3 is mostly equivalent to the original IC3/PDR algorithm [58, 108]. wIC3 differs from the standard IC3 algorithm in a) solver reasoning (SMT solver instead of a SAT solver), b) partition building step that performs equality abstraction, and c) the structure-guided cube generalization step. The procedure first checks whether the property can be trivially violated (lines 13-17), and if not, starts recursively deriving and blocking counterexamples-to-induction (CTI) from the topmost frame (lines 18-31). Given a solver solution s, an abstract state $\hat{s}$ is derived using PartitionBuilder (lines 21, 34). Lines 22, 35 invokes StructGen (Alg. 1) to generalize $\hat{s}$ into a generalized cube c_{gen}. All other parts, including checking for backward reachability using RecBlockCube (lines 32-43), learning a frame clause using BlockCube (lines 44-46), and propagating frame clauses forward using PropagateClauses (lines 47-54), remain identical to the standard incremental induction algorithm.

Extending wIC3+EA with uninterpreted functions is achieved by a) substituting $\mathcal{P}$ with the data-abstracted problem $\bar{\mathcal{P}}$ as a preprocessing step, and b) extending RefineEA with 1-step data refinement checks as detailed in Section 3.4.4.

Algorithm 3 Word-level IC3 with Eqaulity Abstraction

```
procedure wIC3+EA($\mathcal{P}$)                                   ▷ $\mathcal{P} = \langle X, Init, T, P \rangle$
    $F.push(Init)$                                                 ▷ initialize frames with $F_0 = Init$
    $\sigma_{refine} \leftarrow \emptyset$                         ▷ the set of symbols from new terms from REFINEEA
                                                                   ▷ $\mathcal{P}$, $F$, $\sigma_{refine}$ are global data structures
    while $\top$ do
        $Inv, \mathcal{C} \leftarrow$ wIC3()                       ▷ run incremental induction in the word-level EA domain
        if $\mathcal{C} \neq \emptyset$ then                       ▷ if an abstract counterexample is returned
            $\mathcal{C} \leftarrow$ REFINEEA($\mathcal{C}$)       ▷ refine the abstract counterexample
            if $\mathcal{C} \neq \emptyset$ then                   ▷ property is proved unsafe
                return Disproved, $\mathcal{C}$                    ▷ $\mathcal{C}$ is a true counterexample
        else                                                       ▷ else property proved safe
            return Proved, Inv                                     ▷ return the inductive invariant

procedure wIC3()
    if $F.size() = 1$ then                                         ▷ initial checks
        if SAT ? [ $F_0 \wedge \neg P$ ]:  solution $s$ then       ▷ 0-step check
            $\hat{s} \leftarrow$ PARTITIONBUILDER($s$)             ▷ build a partition distribution
            $c_{gen} \leftarrow$ STRUCTGEN($\hat{s}$, $F_0 \wedge \neg P$)
            return $\emptyset$, $\langle c_{gen} \rangle$         ▷ $Init$ is unsafe, return the trivial counterexample
    while $\top$ do
        $n \leftarrow F.size() - 1$                                ▷ $n$ equals index of the topmost frame
        while SAT ? [ $F_n \wedge T \wedge \neg P'$ ]:  solution $s$ do
            $\hat{s} \leftarrow$ PARTITIONBUILDER($s$)             ▷ build a partition distribution
            $c_{gen} \leftarrow$ STRUCTGEN($\hat{s}$, $\neg P$)    ▷ generalize $\hat{s}$ to an abstract cube
            $\mathcal{C}_{partial} \leftarrow \langle \neg P \rangle$    ▷ initialize the partial abstract counterexample
            $\mathcal{C}_{partial}.push(c_{gen})$                  ▷ add $c_{gen}$ to the partial abstract counterexample
            $\mathcal{C}_{full} \leftarrow$ RECBLOCKCUBE($\mathcal{C}_{partial}$, $c_{gen}$, $n - 1$)    ▷ try blocking $c_{gen}$ from $F_n$
            if $\mathcal{C}_{full} \neq \emptyset$ then            ▷ if a full abstract counterexample is returned
                return $\emptyset, \mathcal{C}_{full}$
        $F.push(P)$                                                ▷ add a new frame, $F_{n+1} = P$
        converged $\leftarrow$ PROPAGATECLAUSES()                  ▷ forward propagate frame clauses
        if converged $\neq -1$ then                                ▷ if frames have converged
            return $F_{converged}, \emptyset$                      ▷ return the converged frame as the inductive invariant
    (continued)
```

```
32 procedure RecBlockCube($\mathcal{C}_{partial}$, $c$, $m$)
                                   ▷ $\mathcal{C}_{partial}$ is the partial abstract counterexample
                  ▷ $c$ is an abstract cube that can reach $\neg P$ in $F.size() - m$ steps along $\mathcal{C}_{partial}$
          ▷ In practice, this function is implemented using a priority queue, as described in [108]
33     if SAT ? [ $F_m \wedge T \wedge c'$ ]: solution $s$ then
34         $\hat{s} \leftarrow$ PartitionBuilder($s$)                      ▷ build a partition distribution
35         $c_{gen} \leftarrow$ StructGen($\hat{s}$, $c'$)
36         $\mathcal{C}_{partial}.push(c_{gen})$               ▷ add $c_{gen}$ to the partial abstract counterexample
37         if $m = 0$ then                                  ▷ if reached the initial frame $F_0$
38             return $\mathcal{C}_{partial}$
39         else
40             return RecBlockCube($\mathcal{C}_{partial}$, $c_{gen}$, $m - 1$)        ▷ try blocking $c_{gen}$ from $F_m$
41     else                                          ▷ i.e. $c$ is unreachable from $F_m$ in 1-step
42         BlockCube($c$, $m$)                          ▷ learn a frame clause to restrict $F_{m+1}$
43         return $\emptyset$

44 procedure BlockCube($c$, $i$)
45     $c_{learn} \leftarrow$ MUS($c$, $i$)              ▷ expand $c$ using MUS from $SAT$ ? [ $F_i \wedge T \wedge c'$ ]
46     Add $\neg c_{learn}$ to $F_1 \ldots F_{i+1}$        ▷ restrict frames $1 \ldots i + 1$ by adding clause $\neg c_{learn}$

47 procedure PropagateClauses()
48     for $i = 1$ to $F.size() - 2$ do
49         for each clause $\neg c \in F_i$ do
50             if not SAT ? [ $F_i \wedge T \wedge c'$ ] then
51                 Add $\neg c$ to $F_{i+1}$                  ▷ forward propagate $c$ from frame $i$ to $i + 1$
52         if $F_i = F_{i+1}$ then                                        ▷ if frames converged
53             return $i$
54     return $-1$
```

3.6 Proof of Correctness

Inspired from [58, 108, 79], we list the properties on frames preserved by wIC3+EA (Alg. 3).

(p1) $F_0 = Init$

(p2) $F_i \rightarrow P$

(p3) The clauses F_{i+1} is a subset of F_i for $i > 0$

(p4) $F_i \rightarrow F_{i+1}$

(p5) F_{i+1} is an over-approximation of the image of F_i

(p1-5) are true and preserved by the incremental induction algorithm [58, 108]. More precisely, (p1) is true since F_0 is initialized to *Init* (line 2) and never changed. (p2) is true since each new frame F_i ($i \geq 1$) gets initialized to P (line 28) and since $F_0 \rightarrow P$ due to the initial check (lines 13-17). (p3) is true since PROPAGATECLAUSES and BLOCKCUBE maintain this condition (lines 46 and 51). (p4) is true due to (p2-3) and since $F_0 \rightarrow F_i$ for $i \geq 1$. (p5) is trivially true when entering the loop at line 20 and preserved by PROPAGATECLAUSES. When RECBLOCKCUBE is blocking a cube c from frame F_{m+1} (line 42), this happens when c can reach $\neg P$ in $F.size() - m$ steps. Since there isn't any counterexample of length $\leq F.size()$ (due to (p2)), this means c is unreachable from F_0 in $(F.size() - (F.size() - m)) = m$ steps. Since $F_m \wedge T \wedge c'$ is unsatisfiable, this means c cannot be reached in $m + 1$ steps, preserving (p5). After a refinement iteration, all frame clauses remain valid since the refinement procedure (Alg. 2) is monotonic with respect to the terms describing the EA domain. After each refinement iteration $T_{new} \models T$, implying $F_{new} \models F$, preserving (p1-5).

Lemma 1. *(Correctness) If* wIC3+EA *($\mathcal{P}$) returns an invariant Φ, then Φ is inductive and $\Phi \rightarrow P$ under $\mathcal{P}$.*

Proof. From the IC3 algorithm, let $F_{converged}$ be the frame that reached the fixed point (i.e., $F_{converged} = F_{converged+1}$). Let $\Phi = F_{converged}$. Due to (p5) and (p2), Φ is inductive and $\Phi \rightarrow P$. □

Lemma 2. *(Correctness) If* wIC3+EA *($\mathcal{P}$) returns a counterexample $\mathcal{C}$, then $\mathcal{C}$ has a path under T starting from $Init$ and violating P.*

Proof. Let $\mathcal{C} = \langle c_1, \ldots, c_n \rangle$. By construction, $c_n = \neg P$. From Section 3.4.2, $\mathcal{C}$ is abstractly continuous. The refinement procedure (Alg. 2) will return $\mathcal{C}$ iff $(Init \wedge \big(\bigwedge_{i=1}^{n-1} c_i^i \wedge T^i\big) \wedge c_n^n)$ is satisfiable, implying $\mathcal{C}$ is concretely continuous and a true counterexample. □

Lemma 3. *(Termination)* wIC3+EA *($\mathcal{P}$) will eventually terminate.*

Proof. For a given abstract problem $\hat{\mathcal{P}}$, the IC3 algorithm will eventually terminate since all abstract queries are decidable, the number of abstract states is finite, and the maximum number of frames is bounded by the number of abstract states (due to (p2-5)). Each refinement iteration introduces new term(s) making the abstract state space more precise with respect to the concrete state space. The number of new terms that can be added is limited by the sequential depth of the concrete problem $\mathcal{P}$ (which is finite), making the number of refinement iterations finite. Hence, wIC3+EA will eventually terminate. □

Theorem 3.6.1. wIC3+EA *($\mathcal{P}$) is sound and complete.*

Proof. From Lemmas A.2.1, 2 and 3, wIC3+EA is sound and complete. □

3.7 Evaluation

3.7.1 Setup

We implemented wIC3+EA in C++ in the AVR verifier [121] with a primary focus on model checking of Verilog RTL. AVR uses Yosys [233] as the preprocessor frontend to allow direct translation of Verilog RTL and SystemVerilog assertions (SVA) into a word-level model checking problem. We configured AVR to use Yices 2 [107] version 2.6 for solving all SMT queries during incremental induction in the EA domain and Z3 [97] version 2.5 for all SMT queries during the refinement. We implemented wIC3+EA in AVR with an option for both EA with interpreted functions (i.e., EIF) as well as EA with uninterpreted functions (i.e., EUF). The next chapter presents a detailed description of the internals of AVR, including an independent study of a set og complementary techniques and utilities in AVR for word-level model checking.

We experimented with a total of 535 invariant checking problems (Verilog RTL files with SVA) that can be classified as follows:

- *opensource*: a set of 141 problems collected from benchmark suites accompanying tools *vcegar* [149] (#23), *v2c* [201] (#32) and *verilog2smv* [147] (#86). Problems include cores from picoJava, USB 1.1, CRC generation, Huffman coding, mutual exclusion algorithms, simple microprocessor, etc.

- *industry*: a set of 370 problems collected from industrial collaborators.[3] We inferred these benchmarks to be instances of non-trivial sequential equivalence checks (SEC) between two designs before and after applying several high-level optimizations. Of these, 124 were categorized as *easy* (code sizes between 155 and 761 lines; # of flip-flops between 514 and 931), and 235 as *challenging* (code sizes between 109 and 22065 lines; # of flip-flops between 6 and 7249). The remaining 11 problems involved SEC on a multiplier design before and after clock gating optimization.

- *crafted*: a set of 24 simple problems synthetically created for calibration (includes both control- and data-centric problems).

We compared the following techniques:

From ABC version 1.01 [48]:

- *abc-pdr*: *pdr* is one of the best implementations of the bit-level incremental induction algorithm.

[3] We obtained these designs under non-disclosure agreements and, unfortunately, cannot make their Verilog RTL publicly available. Their word-level description in the BTOR2 format [205] is available at [122].

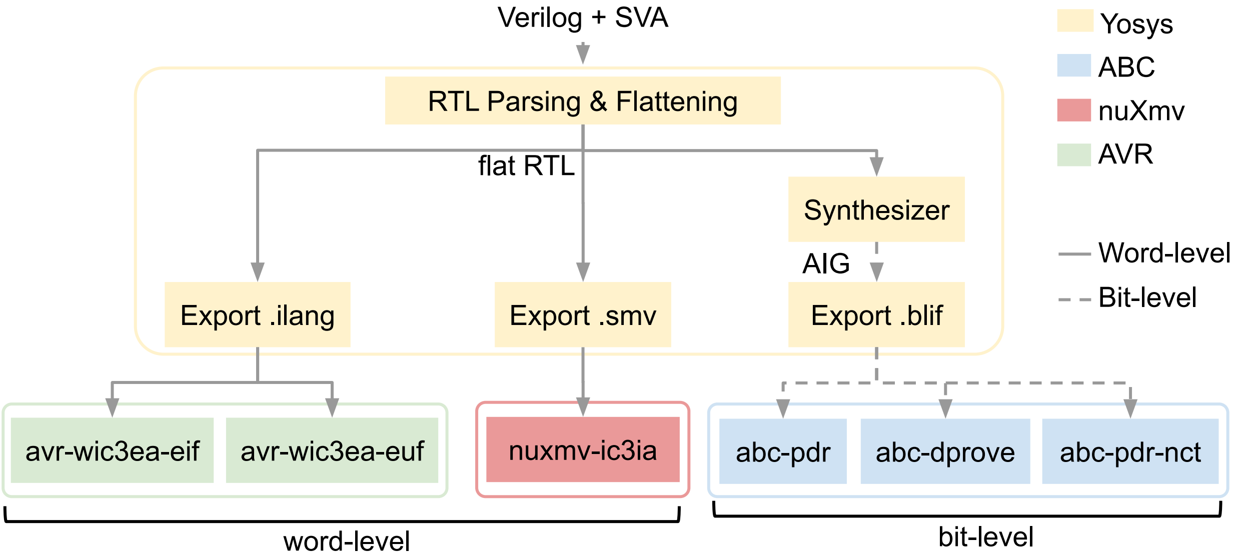


Figure 3.1: Verification toolchain used

- *abc-dprove*: *dprove* employs a preprocessing stage using a portfolio of techniques (BMC, retiming, fraiging, simulation, interpolation, etc.) with carefully-tuned heuristics to quickly solve/reduce the problem. If the problem remains unsolved, *dprove* invokes *pdr* on the reduced problem.
- *abc-pdr-nct*: the -nct flags configure *pdr* to use better generalization [137] and enable *localization* abstraction [143].

From nuXmv version 1.1.1 [69]:

- *nuxmv-ic3ia*: a word-level IC3 implementation in nuXmv using implicit predicate abstraction [79].

From AVR version 2.0 [121]:

- *avr-wic3ea-eif*: wIC3+EA with *interpreted* functions.
- *avr-wic3ea-euf*: wIC3+EA with *uninterpreted* functions.

For comparison against bit-level IC3, we chose implementations from ABC since these have shown exceptional performance in the hardware model checking competition (HWMCC) 2017 [55]. We also considered including other abstraction-based IC3 techniques like *L-IC3* [229], *UFAR* [141] and *PDR-WLA* [142]. However, *PDR-WLA* was not able to process the designs due to input format issues, while *L-IC3* and *UFAR* do not have, to the best of our knowledge, a publicly available implementation. Techniques like [144, 56, 175, 57] do not have implementations that can handle hardware designs.

We set up the experiment as shown in Figure 3.1. Different tools require the input design to be expressed in different file formats and at different levels of abstraction. nuXmv requires input in .smv format, *abc-pdr* and *abc-dprove* operate on synthesized design at the Boolean gate level, while we used the .ilang format[4] for AVR. We used Yosys as the common frontend. The Verilog designs and SVA were parsed by Yosys, which removes any hierarchy and produces flat RTL Verilog. For *nuxmv-ic3ia* and *avr-**, the flat word-level format is syntactically exported by Yosys into the equivalent word-level input formats used by these tools. Since ABC-based tools cannot exploit word-level information and operate at the bit level, we used Yosys to *synthesize* the flat RTL to an And-Inverter Graph (AIG) and exported to ABC in .blif format. All experiments were conducted on a cluster of 163 2.5 GHz Intel Xeon E5-2680v3 processors (cores) running 64-bit Linux. Each verification run was given exclusive access to a single core, with a total memory limit of 16 GB and a time limit of 5 hours.

All experimental data, evaluation scripts, and benchmark files can be retrieved from [123, 122]. The three tool packages used ABC, nuXmv and AVR are publicly available from [1], [17] and [121] respectively.

3.7.2 Results & Discussion

Aggregate Results: Table 3.2 and Figure 3.2 provide an overview on the performance of each tool.[5] Overall, techniques from ABC, nuXmv and AVR solved 480, 389 and 527 problems respectively in total. wIC3+EA with uninterpreted functions (*avr-wic3ea-euf*) performed the best, particularly in the *industry* category. The performance of *avr-wic3ea-eif* is competitive to ABC tools even though ABC tools have a highly tuned and efficient implementation developed over years of innovation.

Tool	**Solved** (535)	**TO**	**MO**	**Error**	**Unique**	**IN** (370)	**OS** (141)	**CR** (24)
abc-pdr	466	69	**0**	**0**	1	308	137	21
abc-dprove	477	57	**0**	1	3	315	**138**	**24**
abc-pdr-nct	466	68	1	**0**	1	308	137	21
nuxmv-ic3ia	389	92	46	8	0	232	133	**24**
avr-wic3ea-eif	461	69	5	**0**	0	302	135	**24**
avr-wic3ea-euf	**526**	**0**	9	**0**	**52**	**368**	134	**24**

Table 3.2: Number of problems solved by each tool.
TO: timed out, MO: out of memory, Unique: solved uniquely (not solved by others)
IN: *industry*, OS: *opensource*, CR: *crafted*

[4] .ilang is a format for textual representation of the Yosys's design.

[5] All plots exclude runs in which a tool reported an error or ran out of memory, and all runtimes refer to CPU time in seconds.

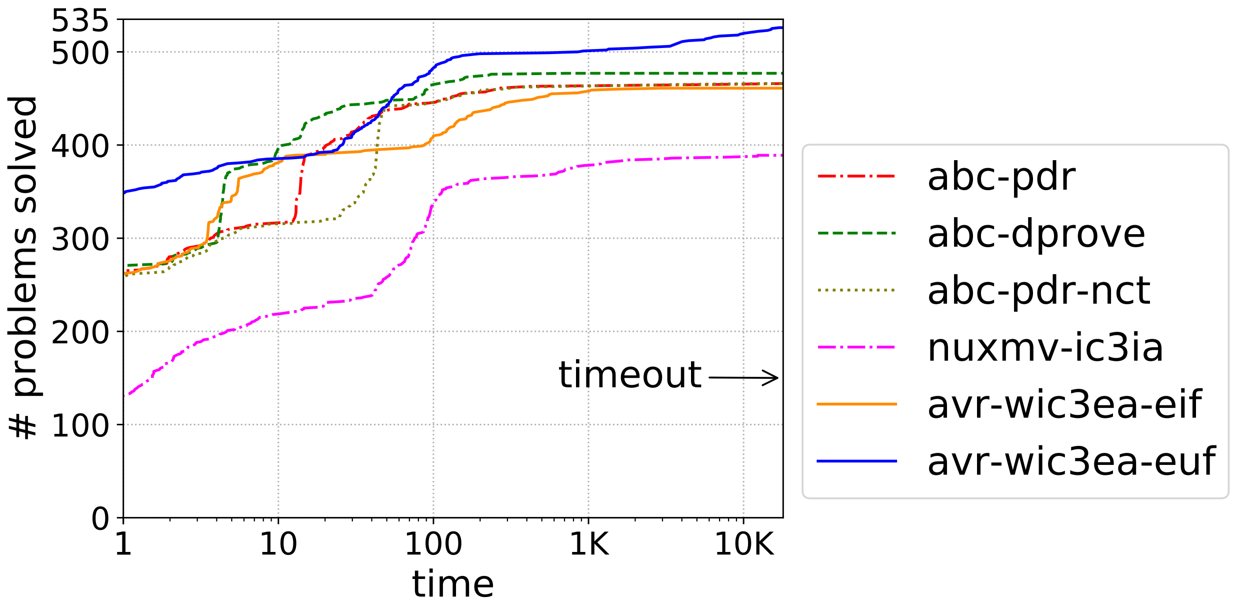


Figure 3.2: Survival plot comparing the number of problems solved versus time

Runtime Comparison: Figure 3.3 compares *avr-wic3ea-eif*'s runtime against other tools. ABC tools marginally dominated *avr-wic3ea-eif*, though there is a significant number where *avr-wic3ea-eif* performed better. Bit-level techniques enjoy the advancements in hardware synthesis that can significantly reduce the complexity in the synthesized design, though they lose this advantage for larger and complex designs. Compared to *nuxmv-ic3ia*, *avr-wic3ea-eif* shows good benefits and demonstrates the benefits of EA over implicit predicate abstraction [79]. Data abstraction using UFs helps *avr-wic3ea-euf* to outperform *avr-wic3ea-eif* in the *industry* category (where the property is control intensive), while *avr-wic3ea-eif* is better in the *opensource* category.

Solver Calls: Figure 3.4 shows the comparison of the total number of solver calls. Bit-level IC3 (represented by *abc-pdr*) makes orders-of-magnitude more SAT solver calls compared to the number of SMT calls made by word-level tools. Even with many more solver calls, bit-level techniques are competitive to word-level techniques with respect to runtime (Figure 3.3), indicating the advancement gap between SAT versus SMT solving. Equality abstraction and structural cube generalization allows *avr-wic3ea-eif* to require fewer solver calls than *nuxmv-ic3ia*. The large number of solver calls made by *avr-wic3ea-euf* compared to *avr-wic3ea-eif* in the *opensource* category reflects the importance of correct abstraction procedure and suggests possible benefits from a hybrid data abstraction using UFs on a subset of data operations beyond an *interpretation threshold* W, that can tune automatically based on the nature of the property. We study this in greater detail in the next chapter.

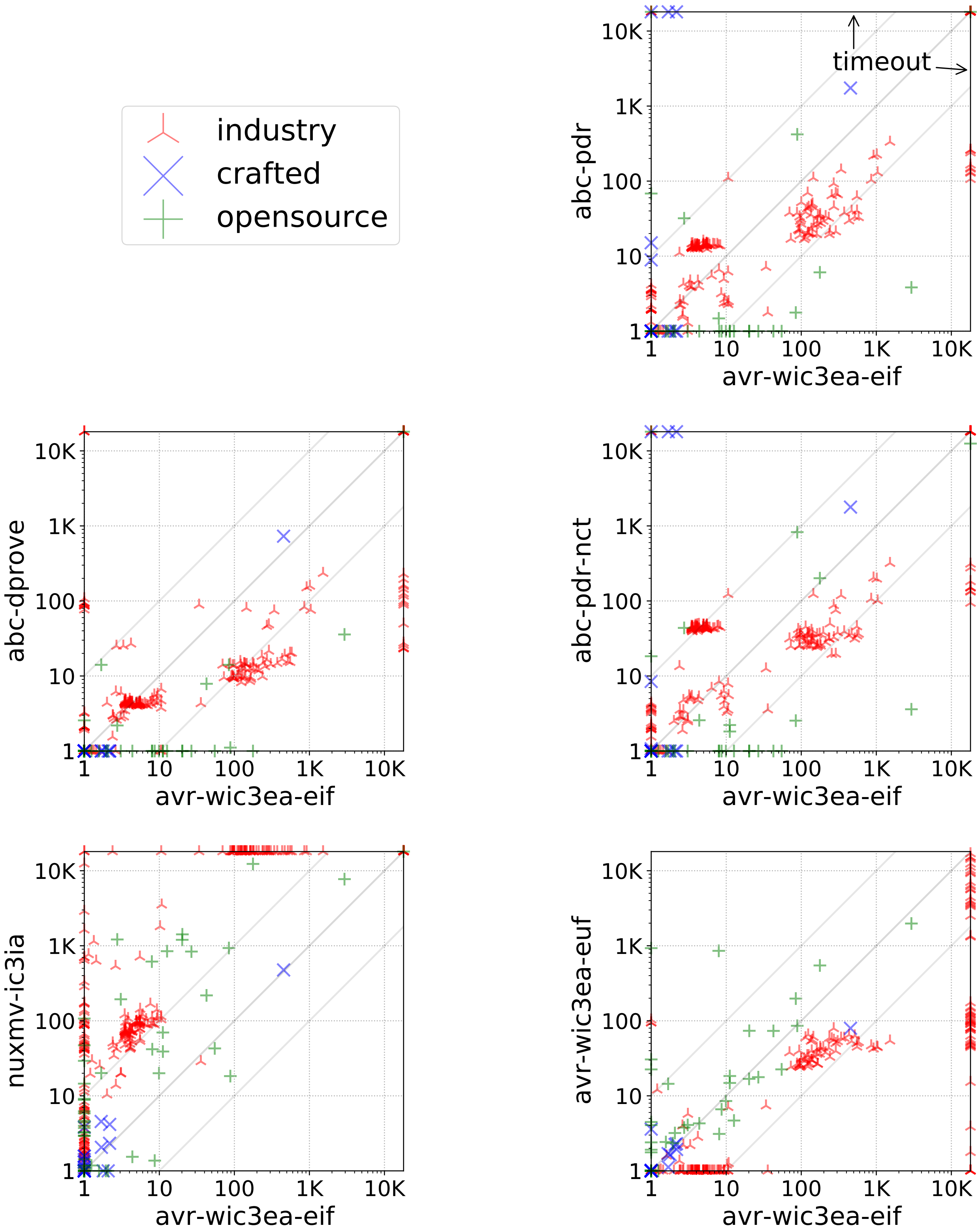


Figure 3.3: *avr-wic3ea-eif* runtime comparisons.
avr-wic3ea-eif is always on x-axis. *avr-wic3ea-eif*'s times are better (resp. worse) above (resp. below) the diagonal.

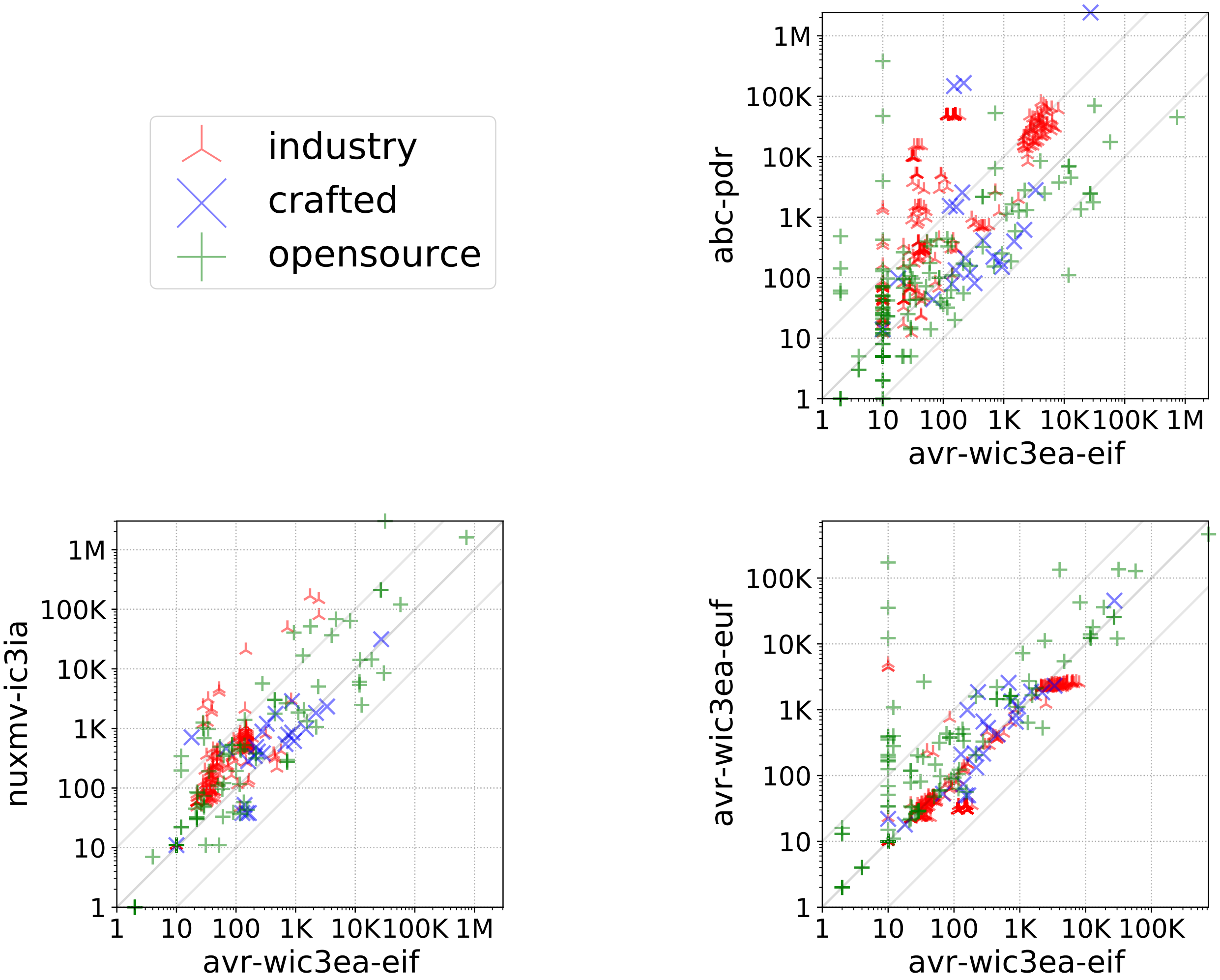


Figure 3.4: Number of solver calls (SAT solver calls for *abc-pdr*, SMT solver calls for others) *avr-wic3ea-eif*'s numbers are better (resp. worse) above (resp. below) the diagonal.

Clause Learning: Figure 3.5.a-b compares the number of frame clauses derived by *avr-wic3ea-eif* versus *abc-pdr* and *nuxmv-ic3ia*. *avr-wic3ea-eif* requires orders-of-magnitude fewer clauses compared to *abc-pdr*, showing the benefits of word-level clause learning as against weak propositional learning. Fewer frame clauses derived by *avr-wic3ea-eif* as compared to *nuxmv-ic3ia* reflects that EA is better in capturing the important details of the problem compared to implicit predicate abstraction.

Invariant Size: Model checking on Verilog RTL instead of post-synthesis Boolean netlist has the additional benefit of producing human-readable word-level inductive invariants. *avr-wic3ea-eif* produces a concise and informative word-level inductive invariant with much fewer clauses than one produced by *abc-pdr* (Figure 3.5.c).

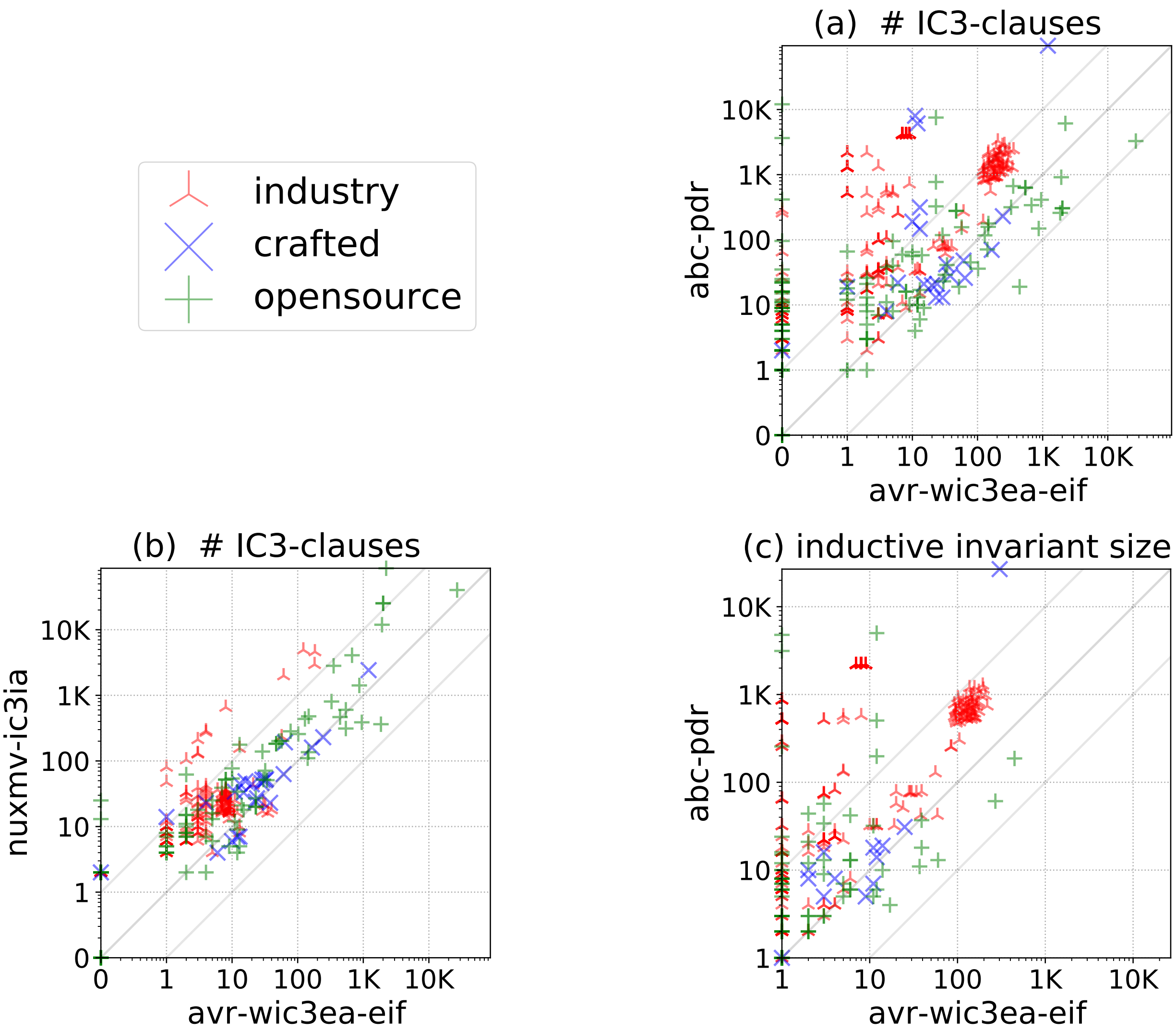


Figure 3.5: IC3 statistics
avr-wic3ea-eif's numbers are better (resp. worse) above (resp. below) the diagonal.

Number of Refinements: Figure 3.6 shows the comparison of the number of refinements required for the techniques that use an abstraction refinement procedure (*abc-pdr-nct*, *nuxmv-ic3ia*, *avr-**). The number of refinements required by *avr-wic3ea-eif* is the least compared to all others, demonstrating the effectiveness of equality abstraction. As expected, *avr-wic3ea-euf* must undergo several refinement iterations for the data-dependent *opensource* category.

3.8 Summary

Equality abstraction suggests an alternative way to raise bit-level IC3 procedure to the word level. EA is implicitly defined by equality relations among the terms in the word-level structure of the

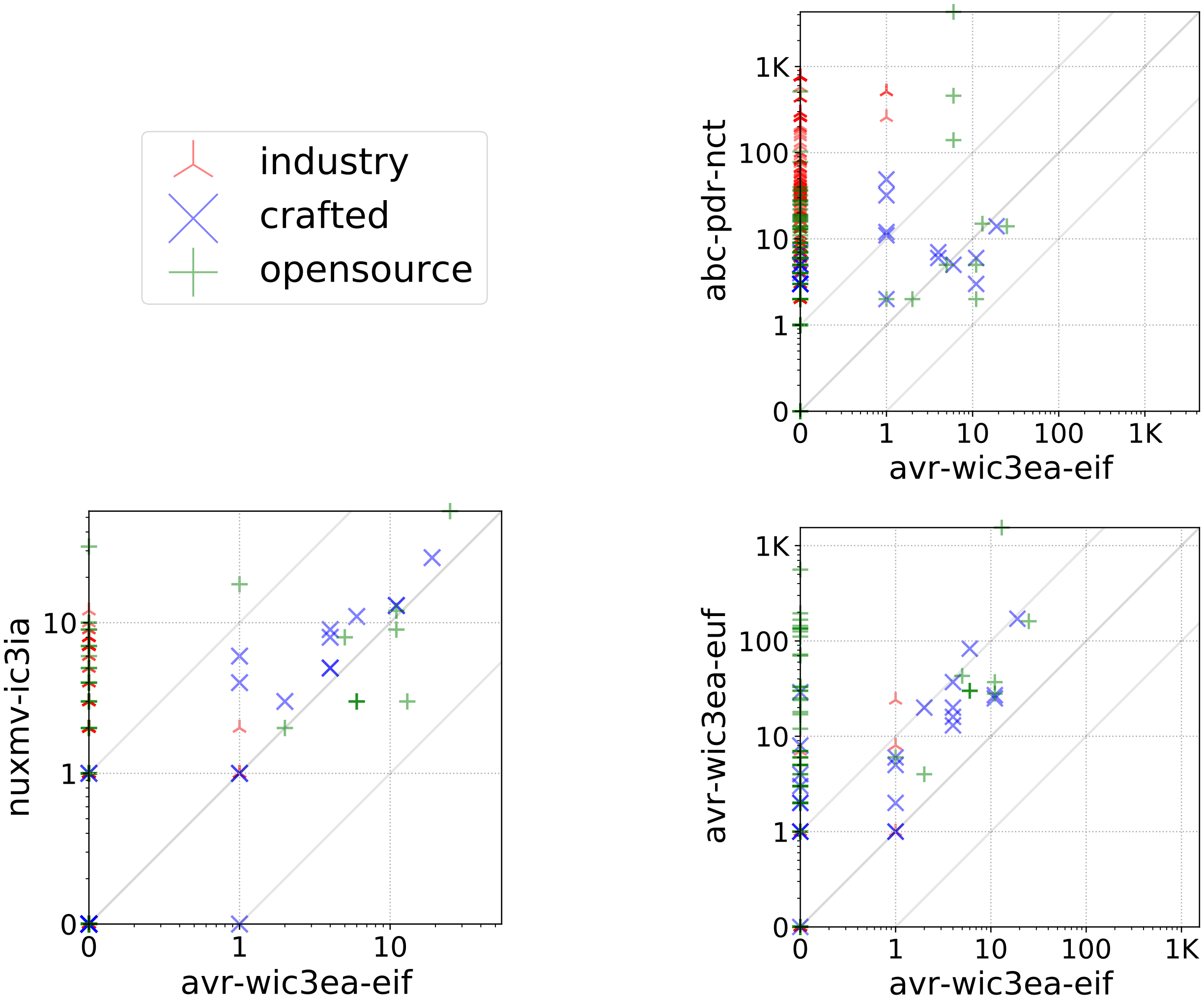


Figure 3.6: Number of refinements
avr-wic3ea-eif's numbers are better (resp. worse) above (resp. below) the diagonal.

problem and offers high granularity. We demonstrate how to integrate incremental induction with EA efficiently and propose a word-level structural cube generalization procedure without any need for additional solver queries or unrolling. The spurious behavior introduced due to the abstraction is incrementally refined through a CEGAR-style technique that refines a spurious abstract counterexample by introducing new terms missing in the original problem description. EA can be additionally combined with data abstraction using uninterpreted functions to give a scalable procedure for data-independent properties. We show the correctness of the technique and evaluate the effectiveness of the approach on a suite of open-source and industrial hardware problems.

CHAPTER 4

Push-button Verification with AVR

In this chapter, we present ***A**bstractly **V**erifying **R**eachability* with **AVR**, a push-button model checker for verifying state transition systems directly at the word level. AVR is designed, primarily, for verifying safety properties of hardware. AVR was independently evaluated to be the best model checker in the Hardware Model Checking Competition (HWMCC) 2020 [52], winning the competition in seven out of nine competition tracks.

AVR's key verification technique is the word-level incremental induction engine using equality abstraction, i.e., WIC3+EA, described in detail in the previous chapter. Additionally, AVR implements a variety of complementary verification techniques, such as hybrid data abstraction, bit-field extraction and concatenation interpreter, proof race, etc., to increase its scalability for different kinds of verification problems. In this chapter, we describe AVR and its different techniques in detail, along with a thorough evaluation of AVR on model checking problems from HWMCC 2020.

The chapter is organized as follows: §4.1 describes the overall architecture of AVR. §4.2 presents different verification techniques in AVR to complement WIC3+EA. §4.3 details the certificates produced by AVR and their applications in guaranteeing provable assurance. §4.4 provides an in-depth evaluation on HWMCC 2020 benchmarks. §4.5 presents case studies that evaluate the usage of AVR for verifying problems beyond hardware, and §4.6 concludes the chapter with a discussion of AVR's strengths and weaknesses.

4.1 System Architecture

Figure 4.1 shows AVR's architecture and verification flow. AVR accepts inputs in different formats and automatically extracts the first-order logic encoding of the transition system to perform word-level verification. Upon termination, AVR either produces a *proof certificate*, in the form of a state formula representing an *inductive invariant*, if the safety property holds, or a counterexample execution trace if it fails.

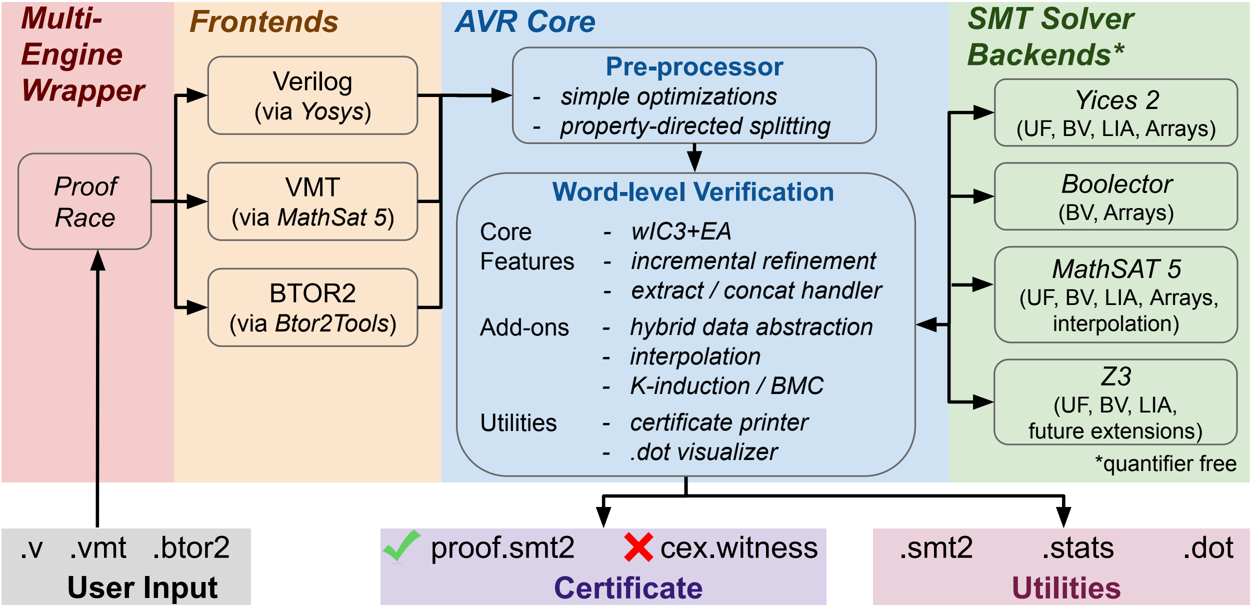


Figure 4.1: Verification flow with AVR
UF: uninterpreted functions, BV: bit-vectors, LIA: linear integer arithmetic

***Frontends*:** automatically extract the word-level model checking problem from inputs in different formats using openly-available tools.

–	Verilog RTL + SVA [28]	using Yosys [233]
–	VMT [26]	using MathSAT 5 [78]
–	BTOR2 [205]	using Btor2Tools [6]

***AVR Core*:** performs word-level verification using a collection of verification techniques, detailed in Section 4.2.

***SMT Solver Backends*:** use the latest versions of state-of-the-art SMT solvers using a C++ API to efficiently integrate incremental solver reasoning with verification techniques. AVR currently integrates:

- Yices 2 [106] with support for quantifier-free reasoning with a combination of uninterpreted functions, bitvectors, linear integer arithmetic and arrays.

- Boolector [204] with support for quantifier-free reasoning with a combination of bitvectors and arrays.

- MathSAT 5 [78] with support for quantifier-free reasoning with a combination of uninterpreted functions, bitvectors, linear integer arithmetic, arrays, and extracting interpolants [195].

- Z3 [97] with support for quantifier-free reasoning with a combination of uninterpreted functions, bitvectors and linear integer arithmetic.

***Multi-Engine Wrapper*:** allows process-level parallelism by running multiple configurations of AVR in parallel using *proof race*, elaborated later in Section 4.2.3.

***Utilities*:** provide useful feedback to the user, including:

- Printing the problem in SMT-LIB format [46].
- Graphical visualizations of the problem and the word-level clause learning.
- Detailed statistics report on the input design and the verification run.

4.2 Techniques

AVR implements WIC3+EA as its primary verification technique. Additionally, AVR also implements a collection of core techniques and extensions to complement WIC3+EA in improving model checking performance and usability for different kinds of verification problems.

4.2.1 Core Techniques

***Pre-processing Optimizations*:** perform simple transformations to standardize and optimize the model checking problem extracted from different input formats. These transformations include:

- Simplifying operations with constant operands through *constant propagation*.
- Removing state variables outside the sequential cone of the safety property through a simple *sequential dependency analysis*.
- Automatic *zero and sign extension* of operands to SMTLIB-compatible versions.

***Partial Interpreter for Extraction and Concatenation*:** adds a novel dedicated handler to add light-weight interpretation of bit-field extraction and concatenation operations. Many hardware systems described at RTL level involve many bit-field extraction and concatenation operations. Abstracting them as uninterpreted can result in too many data refinement lemmas learnt during WIC3+EA with EUF abstraction. These operations are easy to simplify and rather cheap to partially interpret. We implemented a simple procedure that adds partial interpretation of extract/concat operations instead of completely abstracting them as uninterpreted in the EUF logic. For an

Operation	Original form (X)	Simplified form (X_{simple})
Extract	$\{6_3, A_2, B_3\}$[5:2]	$\{ 0_1, A_2, B_3$[2:2] $\}$
Concat	$\{ 6_3, \{ 3_2, A_2 \}, B_3 \}$	$\{ 27_5, A_2, B_3 \}$
Extract	$\{ 6_3, \{ 3_2, A_2 \}, B_3 \}$[8:4]	$\{ 11_4, A_2$[1:1] $\}$

Table 4.1: Examples of original and simplified forms of bit-field extraction and concatenation operations
$W_{\mathtt{N}}$ indicates that the word W is N bits wide

extract/concat operation $\bar{X}$ in the EUF logic, AVR automatically derives its equivalent simplified version $\bar{X}_{simple}$, and whenever $\bar{X}$ appears in an EUF query, AVR adds an additional constraint $(\bar{X} = \bar{X}_{simple})$ to the EUF query. The simplified version $\bar{X}_{simple}$ is derived from $\bar{X}$ by *pushing out* all concatenation operations as much as possible. Table 4.1 provides examples of the original operation X and the corresponding simplified form X_{simple}.

***Incremental Caching*:** allows caching frequently used data structures to speed up incremental SMT solving at the cost of increasing memory usage.

***Other Options*:** include minimizing inductive invariants, adding global assumptions lazily, and exploiting randomness during solving by randomizing the SMT solver seed for each query.

4.2.2 Extensions

Additionally, AVR implements a collection of add-on techniques as extensions that can be configured via command-line options to complement the default wIC3+EA engine.

***Interpretation Threshold* W:** bit width $\leq W$: interpreted $||$ bit width $> W$: uninterpreted
This option allows a user to enable *hybrid data abstraction* within wIC3+EA. All variables and operations with bit widths $\leq W$ are encoded in the BV theory and remain interpreted during word-level IC3 in the EA domain, while all terms with bit widths $> W$ are abstracted as uninterpreted and are encoded in the EUF logic. The interpretation threshold W can range from 1 to the largest bit width in the design. This enables AVR to implement a range of hybrid data abstractions, ranging between EUF abstraction ($W = 1$) where all wide operations are abstracted to uninterpreted, and EIF abstraction ($W =$ *max bit-width*) where all operations remain interpreted.

***Algorithm* A:** wIC3+EA $||$ K-induction $||$ BMC
This option provides the user with alternatives to the default wIC3+EA engine in AVR to instead perform verification with K-induction [221, 100] or Bounded Model Checking (BMC) [54, 82].

K-induction performs model checking with explicit unrolling of the transition relation and is especially suited for data-centric problems with shallow convergence depth. BMC enables quick bug hunting through explicit unrolling of the transition relation, especially suited for shallow bugs.

BV Solver B**:** Yices 2 || Boolector || MathSAT 5 || Z3
This option allows configuring AVR to use different SMT solvers for bitvector reasoning.

Property-directed Wide Variable Splitting S**:** Enable ✓ || Disable ✗
This option adds a preprocessing step in AVR that splits wide variables into multiple smaller bit-width variables at bit-field extraction and concatenation boundaries in a property-directed manner, as elaborated in detail in [33, 176].

Forward Check in IC3 F**:** Enable ✓ || Disable ✗
This option allows enabling the standard 1-step forward CTI check in the incremental induction algorithm, detailed in [108], to enable the exploration of counterexample traces longer than the IC3 frame depth.

CEGAR Interpolation in Refinement I**:** Enable ✓ || Disable ✗
This option allows the usage of Craig interpolants [195] to extract new terms when refining equality abstraction, analogous to the introduction of new predicates in IC3ia [79, 80]. AVR uses MathSAT 5 to derive interpolants from a spurious abstract counterexample.

EA Granularity G**:** *Wires + State Variables* || *State Variables* || *None*
This option provides a methodology to focus on different kinds of equality relations within equality abstraction. We can classify the different relations in EA as: a) Boolean predicates e.g., $u < v$, b) state variable equalities e.g., $u = 1$, and c) wire equalities e.g., $v + 1 = 1$. *Wires + State Variables* is the default EA that includes all these relations and has the highest granularity. *State Variables* ignores all wire equality relations absent in the original problem and limits EA to only Boolean predicates and state variable equalities. *None* even ignores all state variable equality relations absent in the original problem to limit EA to only Boolean predicates.

To better understand this option, recall $\mathcal{P}$ from Example 1 in Section 3.2:

$$\begin{aligned}
&\mathcal{P} = \langle X, \mathit{Init}, T, P \rangle \\
&X = \{u, v\} \\
&\mathit{Init} = (u = 1) \wedge (v = 1) \\
&T = (u' = \mathit{ite}(u < v, u + v, v + 1)) \wedge (v' = v + 1) \\
&P = ((u + v) \neq 1)
\end{aligned}$$

where u and v are k-bit wide. $\mathcal{P}$ has one predicate $u < v$, two state variables u, v, and two wires $u + v, v + 1$. Note that $\mathcal{P}$ contains two state variable equalities $(u = 1)$ and $(v = 1)$ present in the initial state formula *Init*, as well as one wire equality $(u + v = 1)$ present in the safety property formula P. Consider a concrete state $s := (u, v) = (1, 2)$. Its corresponding abstract state description in the EA domain for each of the three options are:

Wires + State variables:

$$\begin{aligned}
\hat{s} &:= \{u < v, \top \,|\, \perp\}_1,\ \{1, u \,|\, v \,|\, u + v, v + 1\}_k \\
\mathit{cube}(\hat{s}) &:= (u < v) \wedge (u = 1) \wedge (u + v = v + 1) \wedge (v \neq 1) \wedge (v \neq u) \\
&\quad \wedge (u + v \neq 1) \wedge (u + v \neq u) \wedge (u + v \neq v) \\
&\quad \wedge (v + 1 \neq 1) \wedge (v + 1 \neq u) \wedge (v + 1 \neq v)
\end{aligned}$$

State variables:

$$\begin{aligned}
\hat{s} &:= \{u < v, \top \,|\, u + v = 1, \perp\}_1,\ \{1, u \,|\, v\}_k \\
\mathit{cube}(\hat{s}) &:= (u < v) \wedge (u + v \neq 1) \wedge (u = 1) \wedge (v \neq 1) \wedge (v \neq u)
\end{aligned}$$

None:

$$\begin{aligned}
\hat{s} &:= \{u < v, u = 1, \top \,|\, u + v = 1, v = 1, \perp\}_1 \\
\mathit{cube}(\hat{s}) &:= (u < v) \wedge (u = 1) \wedge (u + v \neq 1) \wedge (v \neq 1)
\end{aligned}$$

Note the following:

- $\mathit{cube}(\hat{s})$ corresponding to *Wires + State Variables* contains 11 literals, while the ones corresponding to *State Variables* and *None* only contain 5 and 4 literals respectively.
- The wire equality relation $u + v = 1$ is present in $\mathcal{P}$ and is therefore retained in *State Variables*. Similarly, relations $u = 1$, $v = 1$, and $u + v = 1$ are retained in *None* while all

other state variable or wire equalities are not.

- All relations present in $\mathcal{P}$, i.e., $u < v$, $u = 1$, $v = 1$, and $u + v = 1$, are retained in each option to ensure all Boolean formulas and branch conditions can be successfully evaluated during word-level IC3 without any ambiguity with each option.

***Default Configuration*:** AVR uses WIC3+EA with EUF abstraction as its default configuration, as detailed in Table 4.2.

	W	***A***	***B***	***S***	***F***	***I***	***G***
Default	1	WIC3+EA	Yices 2	✓	✗	✗	*State Variables*

Table 4.2: AVR's default configuration
W: Interpretation Threshold, ***A***: Algorithm, ***B***: BV Solver, ***S***: Property-directed Wide Variable Splitting, ***F***: Forward Check in IC3, ***I***: CEGAR Interpolation in Refinement, ***G***: EA Granularity, ✓: Enable, ✗: Disable

4.2.3 Proof Race

With the different extensions described in Section 4.2.2, AVR supports a variety of configurations. Without detailed knowledge of the input problem, it is hard to tell upfront which technique will perform the best. Different configurations are useful to tackle different types of problems, though manually trying different configurations can become tedious for the user. To counter this, AVR offers a multi-engine wrapper called *proof race* that automatically runs multiple instances of AVR with different configurations in parallel and offers process-level parallelism. Given a set of specified resource limits, proof race initiates multiple AVR instances and terminates execution as soon as one of these instances successfully *races* to the result. Such a portfolio-based approach is crucial in practice for fast verification performance since no single technique performs best in all cases [68, 55, 52]. Proof race provides an easy, black-box approach that runs the best techniques implemented in AVR in parallel, without needing the user to understand the internals of AVR.

4.3 Provable Correctness

Once a model checking problem is solved, there can be two possible outcomes: either the property is proved safe, or the property is violated.

If the property holds, AVR's primary technique, i.e., WIC3+EA, produces a word-level inductive invariant represented as a state formula in first-order logic. An inductive invariant represents an approximate fixpoint that establishes the property to be true in *all* executions of the system, and therefore, acts as a *proof certificate*. AVR prints these proof certificates directly in the SMT-LIB format [46], which can further be *independently checked* for their correctness using an external

SMT solver. Since proof certificates are at the word level, they are also *human readable* and much easier to relate to the word-level input directly at the source-code level, as against bit-level invariants which are usually too low level and very hard to understand. Proof certificates find their applications in different ways, like to derive inductive validity cores [118] to gain design insights, to derive assume-guarantee verification conditions [139, 224], to automatically derive helper assertions during multi-property verification [131, 105], or to generalize to quantified domains as elaborated later in the next chapter.

When the property is violated, AVR produces a counterexample trace that establishes how to reach a bad state (a state where the property is false) starting from an initial state. Counterexample traces can be replayed using a witness simulator, like the BTOR2 witness simulator [8]. This allows the designer to debug and pin-point the source of error by analyzing the execution leading to the buggy state.

In both cases, confidence in the verification output is achieved with AVR's capability to produce both, independently checkable proof certificates and counterexample traces, and offer *provable assurance* by guaranteeing the correctness of the verification outcome.

4.4 Hardware Model Checking Competition 2020

AVR participated in the prestigious Hardware Model Checking Competition (HWMCC) 2020 [52], which was the 11th competitive event for hardware model checkers. This section provides a comprehensive analysis of AVR's performance at HWMCC 2020 in Section 4.4.1, along with a detailed analysis on the competition benchmarks along with interesting case studies in Sections 4.4.2 and 4.4.3 respectively.

HWMCC 2020 involved a total of 639 model checking problems distributed into 31 different benchmark families, which were categorized as:

1. BV category consisting of 324 problems distributed into 20 benchmark families.

2. BV + Arrays category consisting of 315 problems distributed into 11 benchmark families.

These benchmarks include a wide variety of verification problems from different sources, including:

- PicoRV32 [18]: a RISC-V CPU that implements the RV32IMC instruction set.

- VexRiscV [24]: a FPGA friendly 32-bit RISC-V CPU implementation.

- Zip CPU [27]: a small, lightweight, RISC CPU.

  - QSPI Flash [21]: A Quad-SPI Flash controller.

  - VGA Sim [25]: A Video display simulator.

  - PonyLink [20]: A single-wire bi-directional chip-to-chip interface for FPGAs.

  - dblclockfft [11]: A configurable generator of pipelined *fast fourier transform* (FFT) cores.

  - Sequential equivalence checking problems on industrial multiplier designs [122].

Eight state-of-the-art model checkers participated in the competition, including:
5 word-level checkers–

- *avr*: AVR [124, 125] competed in the competition as a *proof race* (Section 4.2.3) of a total of 16 word-level configurations (Section 4.2.2), composed of 11 configurations of WIC3+EA, 3 configurations of K-induction, and 2 configurations of BMC.

- *nuxmv*: nuXmv [17, 70] used a portfolio of several verification engines running in parallel, including SAT-based IC3, SMT-based IC3 with implicit abstraction [79], K-induction, and BMC.

- *pono*: Pono [19, 189] used a portfolio of several word-level verification techniques, including BMC, K-induction, interpolation-based [196], and different flavors of IC3 [79, 188].

- *cosa2*: CoSA2 [10, 193] is the precursor of Pono and the winner of the BV + Array track in HWMCC 2019 [51] and runs 4 configurations as in the HWMCC 2019 entry: two BMC configurations, K-induction, and interpolant-based model checking [196].

- *btormc*: BtorMC [7, 205] is a simple bounded model checker, distributed as part of Boolector [204].

3 bit-level checkers–

- *abc*: ABC [1, 60] is the winner of several previous editions of HWMCC, e.g., 2017 [55], and runs a portfolio of different bit-level verification techniques, including BMC, K-induction, and PDR.

- *pdtrav*: PdTrav [67] executes a portfolio of several bit-level techniques, including BMC, BDD-based reachability, k-induction, interpolation, and IC3.

- *avy*: executes AVY [228] and k-AVY [157] in various configurations, as well as BMC and PDR.

Each checker had full access to two Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz CPUs, thus combined 16 cores. For each verification run, a timeout of 1 hour of wall-clock time and a memory limit of 120 GB was used. Ref. [53] provides a detailed summary about the competition.[1]

4.4.1 Competition Results

***Aggregate Results*:** Table 4.3 provides an overview of the competition results. ABC and AVR dominated in the BV category, respectively, solving the most number of unsafe and safe problems. AVR dominated in the BV + Arrays category, solving the most number of problems and significantly outperforming the other tools. Overall, AVR solved the most number of problems (547 out of 639 problems), as well as the most number of uniquely-solved problems (i.e., not solved by anyone else). Out of nine sub-tracks in the competition [52], AVR won the competition with seven gold medals, one silver medal, and one bronze medal.

	Tool	Solved	Unsafe	Safe	TO	MO	UNK	Time (s)	Best	Unique
W	*avr*	257	47	**210**	0	0	67	44345	30	11
	nuxmv	245	47	198	79	0	0	44737	30	2
	pono	142	40	102	165	16	1	35943	11	0
	cosa2	135	40	95	189	0	0	39249	12	0
	btormc	115	40	75	208	0	1	38212	51	1
B	*abc*	**262**	**56**	206	62	0	0	36505	87	6
	pdtrav	245	45	200	76	0	3	74891	14	4
	avy	236	40	196	88	0	0	44055	54	0

a) BV category (324 problems)

	Tool	Solved	Unsafe	Safe	TO	MO	UNK	Time (s)	Best	Unique
W	*avr*	**290**	**19**	**271**	0	0	25	27110	76	12
	nuxmv	255	3	252	60	0	0	54927	2	0
	pono	244	18	226	54	6	11	27845	33	1
	cosa2	238	18	220	63	1	13	96905	9	0
	btormc	218	**19**	199	97	0	0	15611	173	1

b) BV + Arrays category (315 problems)

Table 4.3: HWMCC 2020 competition results: Number of problems solved by each tool.
W: word-level, B: bit-level, TO: timed out, MO: out of memory, UNK: unknown, Time: total wall-clock time (in seconds) for solved problems, Best: the number of problems solved with best runtime among all tools, Unique: the number of problems solved uniquely (not solved by others)

***Runtime Performance*:** Figure 4.2 presents the survival plot comparing the number of problems solved by each tool versus wall-clock time. Out of the 547 problems solved by AVR, 431 problems

[1] We thank Armin Biere, Nils Froleyks, and Mathias Preiner for organizing the HWMCC 2020 competition.

were solved in less than a minute, 76 problems took between 1 to 10 minutes, while only 40 problems took more than 10 minutes to be solved by AVR.

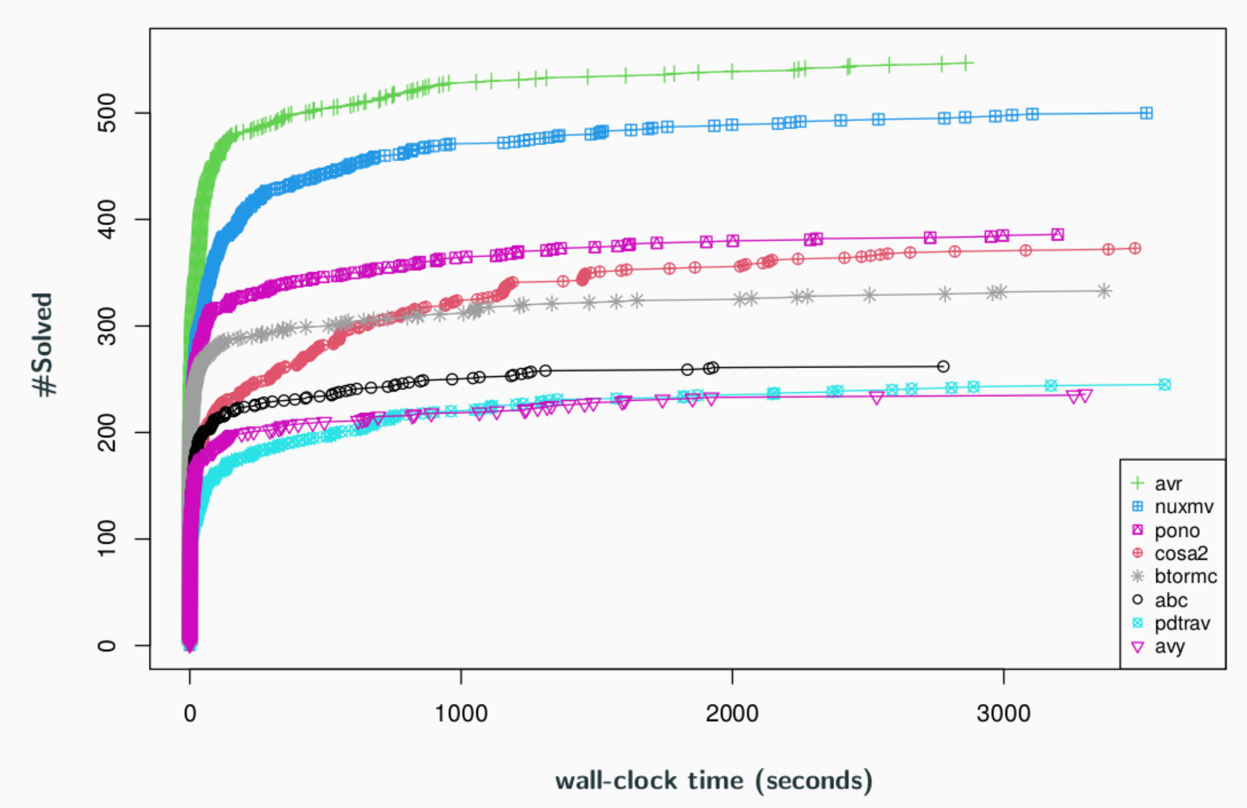


Figure 4.2: HWMCC 2020 competition results: Survival plot comparing the number of problems solved versus runtime

AVR Configurations*:** Table 4.4 lists the 16 configurations of AVR that participated as a *proof race* at the competition. Columns 2-8 (labelled ***W to ***G***) detail command-line options that configure the different AVR extensions (Section 4.2.2). Column 9 (labelled **Best**) lists the number of "wins" during the proof race, i.e., the number of problems solved by a configuration with the best runtime among all configurations during the *proof race*. AVR's default configuration (Table 4.2) won the most number of problems (120 out of 547). The eleven configurations using the WIC3+EA algorithm won on a total of 333 problems, while K-induction and BMC configurations won in the proof race on 170 and 44 problems respectively.

4.4.2 Detailed Analysis

To get a better idea on the different techniques in AVR, we performed a comprehensive analysis of AVR on the HWMCC 2020 problems. We studied the effect of AVR's primary options, namely the

Config.	*W*		*A*	*B*	*S*	*F*	*I*	*G*	Best	
Default	1	(EUF)	wIC3+EA	Yices 2	✓	✗	✗	*SV*	**120**	**333**
2	MAX	(EIF)	wIC3+EA	Boolector	✗	✗	✗	*SV*	47	
3	1	(EUF)	wIC3+EA	Boolector	✗	✗	✗	*SV*	46	
4	8	(hybrid)	wIC3+EA	Boolector	✗	✓	✓	*Wires + SV*	36	
5	4	(hybrid)	wIC3+EA	Yices 2	✓	✓	✓	*SV*	33	
6	1	(EUF)	wIC3+EA	Boolector	✓	✗	✗	*SV*	14	
7	8	(hybrid)	wIC3+EA	Yices 2	✓	✗	✓	*SV*	12	
8	MAX	(EIF)	wIC3+EA	Boolector	✗	✓	✓	*SV*	8	
9	16	(hybrid)	wIC3+EA	Boolector	✓	✗	✓	*SV*	8	
10	1	(EUF)	wIC3+EA	Boolector	✓	✗	✗	*None*	6	
11	32	(hybrid)	wIC3+EA	Boolector	✗	✗	✗	*None*	3	
12	MAX	(BV logic)	K-induction	Boolector	✗	-	-	-	80	170
13	MAX	(BV logic)	K-induction	Yices 2	✓	-	-	-	79	
14	MAX	(BV logic)	K-induction	Boolector	✓	-	-	-	11	
15	MAX	(BV logic)	BMC	Boolector	✓	-	-	-	43	44
16	1	(EUF logic)	BMC	Yices 2	✓	-	-	-	1	

Table 4.4: Different configurations from AVR's proof race from HWMCC 2020
W: Interpretation Threshold, ***A***: Algorithm, ***B***: BV Solver, ***S***: Property-directed Wide Variable Splitting, ***F***: Forward Check in IC3, ***I***: CEGAR Interpolation in Refinement, ***G***: EA Granularity, MAX: max bit width, *SV*: State Variables, ✓: Enable, ✗: Disable, **Best**: number of problems solved with best runtime among all configurations

Algorithm *A* and the Interpretation Threshold *W*, by comparing 10 different AVR configurations with varying values for these options compared to AVR's default configuration. We executed each configuration independently on all 639 HWMCC 2020 benchmarks and used a timeout of 20 minutes and a memory limit of 10 GB for each verification run.

Aggregate Results: Table 4.5 summarizes the results for each configuration. Collectively, these 10 configurations solved 496 out of 639 problems. The wIC3+EA algorithm significantly outperformed K-induction and BMC on safe problems, while BMC dominated on unsafe problems. No clear trend emerged that could show a direct correlation between the number of problems solved and the interpretation threshold *W*, which controls the amount of data abstraction in wIC3+EA. wIC3+EA with EIF abstraction (i.e., with $W =$ *max bit-width*) performed the best, solving the most number of problems.

Benchmark Families: Table 4.6 breaks down the results per benchmark family in each category for the four primary AVR techniques– a) wIC3+EA with EIF abstraction (i.e., $W =$ *max bit-width*), b) wIC3+EA with EUF abstraction (i.e., $W = 1$), c) K-induction, and d) BMC.

Different techniques emerged as better for different benchmark families. For example, wIC3+EA (EUF) dominated in B1 and B2, K-induction dominated in A1, and wIC3+EA (EIF) dominated in A2 and A3. This is attributed to the diverse set of verification problems included in

Algorithm *A*	Interpretation Threshold *W*	Solved	Unsafe	Safe	TO	MO	UNK	Time (s)
wIC3+EA	1 (EUF)	317	17	300	282	4	36	23527
	2 (hybrid)	316	16	300	293	2	28	20075
	4 (hybrid)	313	16	297	282	8	36	24506
	8 (hybrid)	347	20	327	252	2	38	27887
	16 (hybrid)	331	20	311	268	3	37	20048
	32 (hybrid)	337	19	318	264	1	37	18476
	64 (hybrid)	352	18	334	251	2	34	19629
	MAX (EIF)	**361**	19	**342**	241	1	36	20463
K-induction	MAX (BV logic)	283	29	254	335	0	21	10825
BMC	MAX (BV logic)	31	**31**	0	527	38	43	3652

Table 4.5: Number of problems solved with different algorithm *A* and interpretation thresholds *W*
MAX: max bit width, TO: timed out, MO: out of memory, UNK: unknown, Time: total CPU time (in seconds) for solved problems

the competition benchmarks, ranging from data-independent industrial benchmarks (B2, B13, and B16) where EUF abstraction is very effective, to full-fledged RISC cores (B4, A2, A3, A5, A6) where EIF abstraction is most suited, and even data-intensive FFT implementations (A1) where K-induction shows the best performance. Overall, wIC3+EA (EUF) dominated in the BV category, while wIC3+EA (EIF) solved the most number of problems in the BV + Arrays category.

4.4.3 Case Studies

4.4.3.A When wIC3+EA (EUF) is Better

Consider the problem industry/cal210 in the B2 benchmark family. This problem involves sequential equivalence checking between a highly optimized, refactored, and retimed industrial design versus an unoptimized version. All model checkers from the HWMCC 2020 competition (except AVR) failed to solve this problem. The problem has 113 state variables with 1009 total state bits, and contains several wide arithmetic, shift, bitwise, and logical operations, including 21-bit multiplication operations. Out of the 113 state variables, only 16 of them are single bit, i.e., Boolean variables.

Table 4.7 shows that both wIC3+EA (EIF) and K-induction timed out on this problem, while wIC3+EA (EUF) solves the problem in 41 seconds. The property is completely data independent in this problem, resulting in wIC3+EA (EUF) solving it without undergoing any refinement. Comparing the average SMT query time among different AVR techniques is particularly interesting, revealing the benefits of EUF abstraction. Since EUF abstracts away wide data operations as uninterpreted, the query time for wIC3+EA (EUF) is orders of magnitude lower than wIC3+EA (EIF). K-induction showed even worse query time since it relies on explicit unrolling of the transition relation many times, which performs particularly slow in the presence of wide multipliers.

The control-centric nature of the property results in data abstraction to be already precise, re-

		Benchmark Family	#	wIC3+EA (EIF)			wIC3+EA (EUF)			K-induction			BMC
				T	S	U	T	S	U	T	S	U	T/U
BV	B1	2019/wolf/2019C/qspiflash	76	57	57	0	**62**	62	0	9	9	0	0
	B2	2019/goel/industry/cal	44	28	28	0	**40**	40	0	0	0	0	0
	B3	2019/mann/di/unsafe/arbitrated_top	27	1	0	1	0	0	0	**7**	0	7	**7**
	B4	2019/wolf/2018D/zipcpu	24	13	13	0	**19**	19	0	12	12	0	0
	B5	2019/wolf/2019A/picorv32	18	0	0	0	0	0	0	0	0	0	0
	B6	2019/beem	15	2	1	1	2	1	1	**4**	0	4	**4**
	B7	2019/wolf/2019C/vgasim	15	2	2	0	4	4	0	**10**	10	0	0
	B8	2020/mann	15	**11**	6	5	**11**	6	5	10	5	5	5
	B9	2019/wolf/2019C/dspfilters	14	3	3	0	2	2	0	**8**	8	0	0
	B10	2019/goel/opensource	13	**9**	8	1	**9**	8	1	3	2	1	1
	B11	2019/mann/di/unsafe/shift_register	12	**7**	0	7	6	0	6	3	0	3	3
	B12	2019/mann/di/unsafe/circular_pointer	11	3	0	3	2	0	2	**4**	0	4	**4**
	B13	2019/goel/industry/gen	9	**9**	9	0	**9**	9	0	0	0	0	0
	B14	2019/wolf/2018D/picorv32	8	**6**	6	0	**6**	6	0	1	0	1	1
	B15	2019/wolf/2019C/zipversa	8	4	4	0	4	4	0	**6**	6	0	0
	B16	2019/goel/industry/mul	5	1	0	1	**5**	3	2	1	0	1	1
	B17	2019/mann	3	0	0	0	0	0	0	0	0	0	**1**
	B18	2019/wolf/2018D/VexRiscv	3	0	0	0	0	0	0	**2**	2	0	0
	B19	2019/wolf/2019B/marlann	3	0	0	0	0	0	0	**3**	3	0	0
	B20	2019/wolf/2018D/ponylink	1	0	0	0	0	0	0	0	0	0	0
	Sum (BV category)		324	156	137	19	**181**	**164**	17	83	57	26	**27**
BV + Arrays	A1	2019/wolf/2019C/dblclockfft	129	73	73	0	53	53	0	**129**	129	0	0
	A2	2019/wolf/2019C/zipcpu/piped	74	**72**	72	0	49	49	0	42	42	0	0
	A3	2019/wolf/2019C/zipcpu/dcache	63	**51**	51	0	21	21	0	19	19	0	0
	A4	2019/wolf/2019A/picorv32	18	0	0	0	0	0	0	0	0	0	0
	A5	2019/wolf/2018A/zipcpu	9	2	2	0	**6**	6	0	1	1	0	0
	A6	2019/wolf/2018A/picorv32	7	**7**	7	0	**7**	7	0	1	1	0	0
	A7	2019/wolf/2019B/marlann	5	0	0	0	0	0	0	**5**	2	3	3
	A8	2019/mann	3	0	0	0	0	0	0	0	0	0	**1**
	A9	2019/wolf/2018A/VexRiscv	3	0	0	0	0	0	0	**3**	3	0	0
	A10	2020/mann	3	0	0	0	0	0	0	0	0	0	0
	A11	2019/wolf/2018A/ponylink	1	0	0	0	0	0	0	0	0	0	0
	Sum (BV + Arrays category)		315	**205**	**205**	0	136	136	0	200	197	3	**4**
	Sum (all)		639	**361**	**342**	19	317	300	17	283	254	29	**31**

Table 4.6: Comparison of the number of problems solved by different AVR techniques in each benchmark family of HWMCC 2020

#: number of problems in the benchmark family, T: total number of problems solved, S: number of problems proved safe, U: number of problems proved unsafe

sulting in wIC3+EA (EUF) proving the property as safe after exploring 7 frames and deriving a word-level inductive invariant with 194 clauses. Manual investigation of the inductive invariant justified its large size, which involved establishing different conditional equalities between several pairs of state variables/wires of the template $[condition_i \rightarrow (V_{optimized} = V_{unoptimized})]$.

4.4.3.B When wIC3+EA (EIF) is Better

Now consider the problem zipcpu/zipcpu_dcache-p028 in the A3 benchmark family. This problem involves performing checking a designer-specified assertion on the data cache unit of the Zip

Metric	wIC3+EA (EIF)	wIC3+EA (EUF)	K-induction
Time (seconds)	T.O.	**41**	T.O.
Total SMT queries	994	4928	7
Average Query Time (seconds)	0.404	0.007	21.852
Max Frame / Depth	2	7	2
Converged Frame / Depth	N/A	3	N/A
# Learned Frame Clauses	51	304	N/A
Average # Literals in Clauses	4.241	2.571	N/A
# CEGAR Refinements	0	0	N/A
# Clauses in Inductive Invariant	N/A	194	N/A

Table 4.7: Comparison of verification techniques in AVR on industry/cal210 benchmark from B2 family

CPU [27], which is a 5-stage pipelined RISC core. All model checkers from the HWMCC 2020 competition (except AVR) failed to solve this problem. The problem has 1221 state variables with 3568 total state bits, and contains several wide arithmetic, shift, bitwise, logical, and array operations, though no multiplication. Out of the 1221 state variables, 1097 of them are Boolean variables.

Metric	wIC3+EA (EIF)	wIC3+EA (EUF)	K-induction
Time (seconds)	**5**	T.O.	T.O.
Total SMT queries	633	131667	132
Average Query Time (seconds)	0.016	0.003	8.420
Max Frame / Depth	6	7	46
Converged Frame / Depth	5	N/A	N/A
# Learned Frame Clauses	47	527	N/A
Average # Literals in Clauses	5.613	2.489	N/A
# CEGAR Refinements	1	2393	N/A
# Clauses in Inductive Invariant	16	N/A	N/A

Table 4.8: Comparison of verification techniques in AVR on zipcpu/zipcpu_dcache-p028 benchmark from A3 family

Table 4.8 shows that both wIC3+EA (EUF) and K-induction timed out on this problem, while wIC3+EA (EIF) solves the problem in 5 seconds. Here, the property is loosely data dependent and does depend on arithmetic and shift operations to correctly decode the instruction opcode. wIC3+EA (EIF) solves the problem after a single CEGAR refinement, introducing a new concatenated term $\{0, 0, 0, \{0, 1, 1\}, 0\}$, i.e., 7-bit constant 6, derived after sequential unrolling along the abstract counterexample, after which the EIF abstraction becomes precise enough to prove the property as safe. wIC3+EA (EUF), which completely abstracts away all data operations, including all arithmetic and shift operations, failed to solve the problem, and learnt too many weak data refinement lemmas in thousands of CEGAR refinement iterations. K-induction timed out at a depth of 46, while wIC3+EA (EIF) converged after exploring just 6 frames. This shows yet another example of the benefits of incremental induction over K-induction, especially for safe problems that have deep convergence depth. In general, K-induction, however, requires fewer but much more complex SMT queries compared to incremental induction.

Overall, WIC3+EA (EIF) derives a concise inductive invariant with just 16 word-level clauses, each one of which can be easily understood by directly correlating with the input design.

4.4.3.C When K-induction is Better

Consider the problem dblclockfft/butterfly_ck1-p063 in the A1 benchmark family. This problem involves checking the correctness of a general purpose pipelined FFT core [11]. All word-level model checkers from the HWMCC 2020 competition (including AVR) were able to solve this problem through K-induction. The problem has 1903 state variables with 7602 total state bits, and contains some wide arithmetic, bitwise, logical, and array operations, though no multiplication. Out of the 1903 state variables, 1578 of them are Boolean variables.

Metric	WIC3+EA (EIF)	WIC3+EA (EUF)	K-induction
Time (seconds)	T.O.	T.O.	**7**
Total SMT queries	77225	145518	33
Average Query Time (seconds)	0.027	0.006	0.158
Max Frame / Depth	11	11	16
Converged Frame / Depth	N/A	N/A	16
# Learned Frame Clauses	3746	3981	N/A
Average # Literals in Clauses	5.386	6.900	N/A
# CEGAR Refinements	9	1942	N/A
# Clauses in Inductive Invariant	N/A	N/A	N/A

Table 4.9: Comparison of verification techniques in AVR on dblclockfft/butterfly_ck1-p063 benchmark from A1 family

Table 4.9 shows that both WIC3+EA (EIF) and WIC3+EA (EUF) timed out on this problem, while K-induction solves the problem in 7 seconds. As expected, the property here is data intensive with heavy dependence on bit-exact data operations and values. The property has a shallow convergence depth, with K-induction proving the property as safe at a depth of 16. Both WIC3+EA (EIF) and WIC3+EA (EUF) struggled with poor clause learning during incremental induction and learnt more than three thousand frame clauses, finally running out of time after exploring 11 frames. WIC3+EA (EUF) also suffered with thousands of CEGAR refinements, learning weak data refinement lemmas on arithmetic, shift and bitwise operations. The difference in average SMT query time among the three techniques is less drastic in this example, which coupled with the shallow convergence depth, led K-induction to prove the property as safe in just 33 SMT queries. K-induction, however, is incapable of producing an inductive invariant that could act as a proof certificate or provide design insights.

4.5 Beyond Hardware

We also explored using AVR to solve verification problems beyond the hardware domain. All results presented in this section can be replicated from [5].

4.5.1 Apache Buffer Overflow

We consider patched versions of two buffer overflow vulnerabilities [158] from standard modules of the Apache web server [2].

apache-escape-absolute corrects a high severity vulnerability `CVE-2006-3747` [3] that fixes the out-of-bounds buffer overflow exploitation which allows a remote attacker to cause a denial of service and execute arbitrary code via crafted URLs. The patched version corrects a check $(c < \texttt{TOKEN\_SZ})$ to $(c < \texttt{TOKEN\_SZ} - 1)$.

apache-get-tag fixes a medium severity vulnerability `CVE-2004-0940` [4] that exploits a buffer overflow when copying user-supplied tag strings into finite buffers. A local attacker may leverage this issue to execute arbitrary code on the affected computer with the privileges of the affected Apache server. The patched version corrects a check that validates the length of the tag strings.

In less than a minute, AVR successfully verifies that both buffer overflow exploits are unreachable in the patched versions for *any* buffer size. AVR also provides human-readable proof certificates that are externally verified using Z3 and provides provable assurance against these security vulnerabilities.

4.5.2 Public Key Authentication Protocol

The Needham-Schroeder public key authentication protocol [203] allows establishing mutual authentication between an initiator A and a responder B, after which some session involving the exchange of messages between them can take place. Unfortunately, this protocol is vulnerable to a man-in-the-middle attack [184]. If an intruder I can persuade A to initiate a session with him, he can relay the messages to B and convince B that he is communicating with A.

We consider an instance of the protocol from HWMCC'19 [50, 212] with 3 initiators and responders each, and with an unsafe state defined as a responder being finished authentication with the intruder as a party. Within a minute, AVR finds an execution trace that establishes how to reach an unsafe state. The counterexample witness produced by AVR can be replayed using the BtorSIM simulator [8] to verify the execution trace and to debug the protocol.

4.6 Summary

AVR provides a variety of techniques to efficiently perform automatic and scalable word-level verification using SMT solvers with provable correctness guarantees. We conclude this chapter with a summary of the strengths and limitations of AVR.

4.6.1 Strengths

Control-centric properties, where much of the complexity lies in the control logic (such as sequential equivalence checking, microprocessor instruction control unit, key-value store) are much easier to verify using AVR. Equality abstraction provides an automatic way to separate important control-flow details from the irrelevant data component. This, combined with data abstraction using EUF abstraction, allows for scalable model checking with the capacity to scale independently of the variable bit widths [120, 124, 52].

Push-button verification using AVR eliminates the need for tedious human intervention in verification such as manual identification of abstraction predicates, manually adding helper assertions, using WIC3+EA coupled with a variety of fully-automatic extensions.

Provable assurance on the verification outcome is guaranteed by AVR using independently checkable proof certificates and counterexample traces.

Useful utilities that AVR provides, such as support for multiple input formats, efficient integration with state-of-the-art SMT solvers, proof race, high-level system statistics, graphical visualizations, etc. contribute to a user-friendly experience and ease of use.

4.6.2 Limitations

Heavy data dependency can make word-level techniques in AVR ineffective for certain problems, especially when most bit-precise values in the data domain play an important role (for example, puzzle solving problems like Tower of Hanoi [155], Peg Solitaire [150], etc. formulated as reachability problems [212]). Logic synthesis and bit-level optimizations [48, 199] can be very useful for such problems and help bit-level checkers perform better than word-level techniques by significantly decreasing the problem complexity at the bit level.

First-order logic fragments beyond quantifier-free bit-vectors, arrays and uninterpreted functions (such as non-linear arithmetic, floating-point numbers, quantifiers, etc.) and properties beyond safety (such as *liveness* and *fairness*) have limited support in the current tool implementation. AVR's primary focus has been on verification of safety properties defined on hardware systems.

CHAPTER 5

Automatic Verification of Distributed Protocols

In this and the following chapter, we propose a pragmatic problem-solving approach for automatically deriving quantified inductive invariants in first-order logic that serve as proof certificates for the *safety* of distributed protocol specifications. In an initial attempt, we explored the ability of AVR to automatically generate (finite) inductive invariants with the intuition that it should be possible to verify an unbounded protocol by analyzing a relatively small finite instance. In collaboration with experts in the distributed systems domain, this led to the development of I4, a proof-of-concept experiment that infers universally-quantified inductive invariants for distributed protocols by generalizing finite-domain inductive invariants automatically generated with AVR.[1] A closer examination of how I4 was generating the invariants through AVR revealed that it lacked a high-level understanding of the transition relation it was analyzing. Recognizing different structural features in distributed protocols, we embarked on developing a completely new protocol verifier called IC3PO, where we introduced simple extensions to the finite-domain incremental induction algorithm to automatically infer the required quantified inductive invariant for a distributed protocol.

In this chapter, we present the key ideas behind IC3PO that enable fully-automatic verification of distributed protocols. We propose *symmetry boosting* to take advantage of a protocol's *spatial* regularity to automatically infer quantified strengthening assertions that reflect the protocol's structural symmetries by exploiting the connection between symmetry and quantification. We will also discuss a novel *finite convergence* procedure to automatically find a minimal finite size, the *cutoff*, that yields a quantified inductive invariant proving safety for any size. We will continue this discussion in the following chapter, where we describe *range boosting* and *hierarchical strengthening* to take advantage, respectively, of a protocol's *temporal* regularity and *hierarchical* structure.

The chapter is organized as follows: §5.1 presents an introduction and §5.2 presents the high-level description of our initial attempt. The rest of the sections describe IC3PO in detail. §5.3

[1] I4 is joint work with Haojun Ma, Jean-Baptiste Jeannin, Manos Kapritsos, and Baris Kasikci. The dissertation author developed the finite model checking support in this work, while other parts were developed by I4's primary contributor Haojun Ma.

presents preliminaries and §5.4 formalizes protocol symmetries. The next three sections detail symmetry boosting during incremental induction (§5.5), relating symmetry to quantification (§5.6), and checking for convergence (§5.7). §5.8 describes simple enhancements to the IC3PO algorithm presented in §5.9. §5.10 describes our experimental evaluation, §5.11 presents a case study, and §5.12 concludes the chapter with a summary.

5.1 Introduction

Our focus in this chapter is on *parameterized verification*, specifically proving *safety* properties of distributed systems, such as protocols that are often modeled above the code level (e.g., [168, 210]), consisting of arbitrary numbers of *identical* components that are instances of a small set of different *sorts*. For example, the client server protocol $CS(i,j)$ from Figure 1.2 is a two-sort parameterized system with parameters $i \geq 1$ and $j \geq 1$ denoting, respectively, the number of clients and servers. Protocol correctness proofs are critical for establishing the correctness of actual system implementations in established methodologies such as [138, 232]. Proving safety properties for such systems requires the derivation of inductive invariants that are expressed as state predicates quantified over the system parameters. Notwithstanding the undecidability result of Apt and Kozen [35],[2] many efforts to find mechanical solutions to the problem have been reported with the pace increasing in recent years. However, the derivation of inductive invariants for distributed protocols continues to be mostly carried out using interactive theorem provers [43, 206, 32] that require significant manual effort and deep domain expertise.

Our proposed solutions are best understood by briefly reviewing earlier efforts. Initially, the pressing issue was the inevitable *state explosion* when verifying a finite, but large, parameterized system [110, 207, 216, 119, 222, 42]. Thus, instead of verifying the "full" system, these approaches verified its *symmetry-reduced quotient,* mostly using BDD-based symbolic image computation [64, 65, 194]. The Murφ verifier [207] was a notable exception in that it a) generated a C++ program that enumerated the system's symmetry-reduced reachable states, and b) allowed for the verification of unbounded systems by taking advantage of *data saturation* which happens when the size of the symmetry-reduced reachable states become constant regardless of system size.

The idea that an unbounded *symmetric* system can, under certain data-independence assumptions, be verified by analyzing small finite instances evolved into the approach of verification by *invisible invariants* [215, 36, 237, 39, 104]. In this approach, assuming they exist, inductive invariants that are universally-quantified over the system parameters are automatically derived by analyzing instances of the system up to a *cutoff* size N_0 using a combination of symbolic reach-

[2] An interesting aside is this quote by Ed Clarke in [86]: "In any case, it is clear that the claim in [35] regarding the infeasibility of automatically checking the correctness of programs with many processes is unduly pessimistic."

ability and symmetry-based abstraction. Noting that an invariant is an over-approximation of the reachable states, the restriction to universal quantification may fail in some cases, rendering the approach incomplete. The invisible invariant verifier IIV [39] employs some heuristics to derive invariants that use combinations of universal and existential quantifiers, but as pointed out in [202], it may still fail and is not guaranteed to be complete.

Instead of manually and painstakingly identifying an inductive invariant, we can draw inspiration from the inductive invariants of small, finite instances of these protocols. The I4 system [186, 187, 185] is premised on the conjecture that a universal inductive invariant for a distributed protocol can be inferred by analyzing a small finite instance using finite-state methods. AVR was adapted in I4 to produce a quantifier-free inductive invariant for a small finite instance of an unbounded protocol that are subsequently generalized with universal quantification, in analogy with the invisible invariants approach, to arbitrary sizes. However, the resulting assertions tended to be quite large, and the approach was also incomplete due to the restriction to universal quantification.

Rather than search for an invariant with a prescribed quantifier prefix, we developed a novel general-purpose protocol verification tool. IC3PO constructively *discovers* the required quantified assertions by performing *symmetric incremental induction* and analyzing the symmetry patterns in learned clauses to infer the corresponding quantifier prefix. By understanding the spatial regularity in distributed protocols, we discovered a connection between symmetry and quantification, namely that they are complementary ways of expressing invariance. IC3PO also implements an algorithmic *finite convergence* procedure to check if the current instance size has captured all possible protocol behaviors and, if not, to systematically increase the finite instance size until protocol behavior saturates and the cutoff size is reached.

5.2 Invariant Generation from a Finite Instance

The key insight of our approach is that the behavior of most distributed protocols does not fundamentally change as their size increases. This insight led us to ask the question: *is it possible to infer the inductive invariant of a distributed protocol by observing a small instance of the protocol?* Our exploration demonstrates that the answer to this question is typically “yes”. Distributed protocols exhibit a high degree of regularity: what is true for a small instance of four nodes is also true for a large instance of 1000 nodes. Let’s understand this regularity using the client server protocol from Figure 1.2 with N servers and M clients. Consider the inductive invariants of small instances of this protocol. The smallest non-trivial instance consists of one server and two clients. The inductive invariant to prove the correctness of that instance is:

$$\begin{array}{|ll|}\hline \neg(\mathit{semaphore}(\mathtt{s}_1) \land \mathit{link}(\mathtt{c}_1, \mathtt{s}_1)) & \land \\ \neg(\mathit{semaphore}(\mathtt{s}_1) \land \mathit{link}(\mathtt{c}_2, \mathtt{s}_1)) & \land \\ \mathit{Safety\ Property} & \\ \hline \end{array}$$

If we consider a larger instance with more clients, say four, the inductive invariant becomes bigger, but remains essentially the same:

$$\begin{array}{|ll|}\hline \neg(\mathit{semaphore}(\mathtt{s}_1) \land \mathit{link}(\mathtt{c}_1, \mathtt{s}_1)) & \land \\ \neg(\mathit{semaphore}(\mathtt{s}_1) \land \mathit{link}(\mathtt{c}_2, \mathtt{s}_1)) & \land \\ \neg(\mathit{semaphore}(\mathtt{s}_1) \land \mathit{link}(\mathtt{c}_3, \mathtt{s}_1)) & \land \\ \neg(\mathit{semaphore}(\mathtt{s}_1) \land \mathit{link}(\mathtt{c}_4, \mathtt{s}_1)) & \land \\ \mathit{Safety\ Property} & \\ \hline \end{array}$$

When we further consider instances with multiple servers, the invariant again increases in size *but not in complexity*. For example, the inductive invariant for an instance with two servers and four clients is:

$$\begin{array}{|ll|}\hline \neg(\mathit{semaphore}(\mathtt{s}_1) \land \mathit{link}(\mathtt{c}_1, \mathtt{s}_1)) & \land \\ \neg(\mathit{semaphore}(\mathtt{s}_1) \land \mathit{link}(\mathtt{c}_2, \mathtt{s}_1)) & \land \\ \neg(\mathit{semaphore}(\mathtt{s}_1) \land \mathit{link}(\mathtt{c}_3, \mathtt{s}_1)) & \land \\ \neg(\mathit{semaphore}(\mathtt{s}_1) \land \mathit{link}(\mathtt{c}_4, \mathtt{s}_1)) & \land \\ \neg(\mathit{semaphore}(\mathtt{s}_2) \land \mathit{link}(\mathtt{c}_1, \mathtt{s}_2)) & \land \\ \neg(\mathit{semaphore}(\mathtt{s}_2) \land \mathit{link}(\mathtt{c}_2, \mathtt{s}_2)) & \land \\ \neg(\mathit{semaphore}(\mathtt{s}_2) \land \mathit{link}(\mathtt{c}_3, \mathtt{s}_2)) & \land \\ \neg(\mathit{semaphore}(\mathtt{s}_2) \land \mathit{link}(\mathtt{c}_4, \mathtt{s}_2)) & \land \\ \mathit{Safety\ Property} & \\ \hline \end{array}$$

Given the above instances, it does not take much ingenuity to manually come up with an inductive invariant that works for *all* instances of this protocol:

$$\begin{array}{|ll|}\hline \forall C \in \mathtt{client}, S \in \mathtt{server} :\ \neg(\mathit{semaphore}(S) \land \mathit{link}(C, S)) & \land \\ \mathit{Safety\ Property} & \\ \hline \end{array}$$

Of course, this protocol is rather simple, and its inductive invariant is quite small. But the principle still applies to more complicated protocols: we can use the inductive invariant of a small instance to infer a *generalized* inductive invariant that works for all instances of the protocol.

5.2.1 Proof of Concept: Utilizing AVR to verify Distributed Protocols

In collaboration with experts in the distributed systems domain, this led to the development of a technique called *Incremental Inference of Inductive Invariants* (I4), that combines the ability of AVR to automatically generate (finite) inductive invariants and the intuition that it should be possible to verify an unbounded protocol by analyzing a relatively small finite instance. In some sense I4 can be viewed, at a high level, as the logical successor of the invisible invariant approach: it replaced the BDD-based projection and generalization procedure by AVR's ability to automatically produce a finite inductive invariant but used Ivy's unbounded check to verify its inductiveness for all sizes.

The reader is referred to [186] for a complete description of I4. Here, we limit our presentation to a brief summary. The I4 system reads an Ivy description of a protocol and creates a small finite instance of it based on a user-supplied size. This was then converted into a logic circuit which was fed to AVR. If the verification failed, AVR produced a counterexample which could be used to debug the protocol description. Otherwise, AVR produced a finite inductive invariant for the given size which was passed to a generalization procedure that produced a corresponding *universally-quantified* invariant. This step was quite ad hoc since it presumed that universal quantification is the "correct" generalization of the finite invariant. In any case, this quantified invariant was then passed to the Ivy verifier to determine if it was inductive in the unbounded case. If it was, verification succeeded. If not, the next step depended on whether the property itself or the assertion produced by AVR was the reason for failure. Failure due to the property was handled by manually increasing the instance size and repeating the process. Failure due to one or more AVR-generated assertions was handled by strengthening the assertions in another ad hoc refinement step followed by re-running the Ivy unbounded check.

5.2.2 Limitations of I4

Our verdict on this initial exercise was that AVR's ability to automatically generate finite inductive invariants looked promising and should be explored further. Tested on a wide variety of protocols that involved finite-state processes, it succeeded in solving 13 out of 29 protocols in times ranging from a few seconds to about 20 minutes, as shown in Table 5.1. However, it failed in various unexpected ways, and the invariants it produced were quite verbose and hard to understand. A closer examination of how AVR was generating the invariants revealed that it lacked a high-level understanding of the transition relation it was analyzing and was learning too many specific facts that were hard to generalize into compact invariants.

The most obvious structural feature of distributed protocols is *symmetry*. We decided to embrace these symmetries and use them to let the incremental induction algorithm know it was searching in a structurally symmetric state space. With this central idea, we embarked on developing another prototype, called IC3PO, which we describe in detail in the rest of this chapter.

	Human	I4			UPDR		
Protocol (#29)	Inv	Time	Inv	SMT	Time	Inv	SMT
tla-consensus	1	4	1	7	0	1	38
tla-tcommit	3	unknown		71	1	3	214
i4-lock-server	2	2	2	35	1	2	133
ex-quorum-leader-election	3	32	14	15429	11	3	1007
pyv-toy-consensus-forall	4	unknown		5949	10	3	590
tla-simple	8	4	3	1319	timeout		
ex-lockserv-automaton	2	3	15	1731	21	9	3855
tla-simpleregular	9	unknown		14787	timeout		
pyv-sharded-kv	5	4	15	2101	6	7	784
pyv-lockserv	9	3	15	1606	14	9	3108
tla-twophase	12	unknown		10505	67	14	12031
i4-learning-switch	8	22	11	26345	timeout		
ex-simple-decentralized-lock	5	14	22	5561	4	2	677
i4-two-phase-commit	11	4	16	4045	16	9	2799
pyv-consensus-wo-decide	5	1144	42	41137	100	4	8563
pyv-consensus-forall	7	1006	44	156838	490	6	24947
pyv-learning-switch	8	387	49	51021	278	11	3210
i4-chord-ring-maintenance	18	timeout			timeout		
pyv-sharded-kv-no-lost-keys	2	unknown		1232	unknown		73
ex-naive-consensus	4	unknown		15141	unknown		1325
pyv-client-server-ae	2	unknown		1483	unknown		132
ex-simple-election	3	unknown		2747	unknown		1147
pyv-toy-consensus-epr	4	unknown		5944	unknown		473
ex-toy-consensus	3	unknown		2797	unknown		348
pyv-client-server-db-ae	5	unknown		81509	unknown		422
pyv-hybrid-reliable-broadcast	8	unknown		34764	unknown		713
pyv-firewall	2	unknown		344	unknown		130
ex-majorityset-leader-election	5	error			unknown		2350
pyv-consensus-epr	7	unknown		177189	unknown		7559
No. of problems solved	(out of 29)	13			14		

Table 5.1: Comparison of I4 against UPDR on a variety of distributed protocols
Time: run time (seconds), Inv: # assertions in inductive proof, SMT: # SMT queries

5.3 Preliminaries

Algorithm 3 describes a toy consensus protocol from [23] in the TLA+ language [168].[3] The protocol has three named sorts $S = [\texttt{node}, \texttt{quorum}, \texttt{value}]$ introduced by the CONSTANTS declaration, and two relations $R = \{vote, decision\}$, introduced by the VARIABLES declaration, that

[3] The description in [23] is in the Ivy [210] language and encodes set operations in relational form with a $member$ relation representing $\in$.

Algorithm 3 Toy consensus protocol in the TLA+ language

```
1  CONSTANTS node, quorum, value
2  VARIABLES $vote$, $decision$
3  $vote \in (\texttt{node} \times \texttt{value}) \rightarrow \text{BOOLEAN}$
4  $decision \in \texttt{value} \rightarrow \text{BOOLEAN}$
5  ASSUME $\forall Q \in \texttt{quorum} : Q \subseteq \texttt{node} \ \wedge \ \forall Q_1, Q_2 \in \texttt{quorum} : Q_1 \cap Q_2 \neq \{\}$
6  $didNotVote(n) \triangleq \forall V \in \texttt{value} : \neg vote(n, V)$
7  $chosenAt(q, v) \triangleq \forall N \in q : vote(N, v)$
8  $CastVote(n, v) \triangleq didNotVote(n) \ \wedge \ vote' = [vote \ \text{EXCEPT} \ ![n, v] = \text{TRUE}]$
                                 $\wedge \ \text{UNCHANGED} \ decision$
9  $Decide(q, v) \triangleq chosenAt(q, v) \ \wedge \ decision' = [decision \ \text{EXCEPT} \ ![v] = \text{TRUE}]$
                                 $\wedge \ \text{UNCHANGED} \ vote$
10 $Init \triangleq \forall N \in \texttt{node}, V \in \texttt{value} : \neg vote(N, V) \ \wedge \ \forall V \in \texttt{value} : \neg decision(V)$
11 $T \triangleq \exists N \in \texttt{node}, Q \in \texttt{quorum}, V \in \texttt{value} : CastVote(N, V) \vee Decide(Q, V)$
12 $P \triangleq \forall V_1, V_2 \in \texttt{value} : decision(V_1) \wedge decision(V_2) \Rightarrow V_1 = V_2$
```

are defined on these sorts. Each of the sorts is understood to represent an unbounded domain of distinct elements with the relations serving as the protocol's state variables. The global axiom (line 5) defines the elements of the quorum sort to be subsets of the node sort and restricts them further by requiring them to be pair-wise non-disjoint. We will refer to node (resp. quorum) as an *independent* (resp. *dependent*) sort. The protocol transitions are specified by the actions $CastVote$ and $Decide$ (lines 8-9) which are expressed using the current- and next-state variables as well as the definitions $didNotVote$ and $chosenAt$ (lines 6-7) which serve as *auxiliary non-state* variables. Lines 10-12 specify the protocol's initial states, transition relation, and safety property.

Viewed as a parameterized system, the *template* of an arbitrary n-sort distributed protocol $\mathcal{P}$ will be expressed as $\mathcal{P}(\mathtt{s}_1, \ldots, \mathtt{s}_\mathtt{n})$ where $S = [\mathtt{s}_1, \ldots, \mathtt{s}_\mathtt{n}]$ is an ordered list of its sorts, each of which is assumed to be an unbounded uninterpreted set of distinct *constants*. As a mathematical transition system, $\mathcal{P}$ is defined by a) its state variables which are expressed as k-ary relations on its sorts, and b) its actions which capture its state transitions. We also note that non-Boolean functions/variables can be easily accommodated by encoding them in relational form, e.g., $f(\mathtt{x}_1, \mathtt{x}_2, \ldots) = \mathtt{y}$. We will use $Init$, T, and P to denote, respectively, a protocol's initial states, its transition relation, and a safety property that is required to hold on all reachable states. A finite instance of $\mathcal{P}$ will be denoted as $\mathcal{P}(|\mathtt{s}_1|, \ldots, |\mathtt{s}_\mathtt{n}|)$ where each named sort is replaced by its finite size in the instance. Similarly, $Init(|\mathtt{s}_1|, \ldots, |\mathtt{s}_\mathtt{n}|)$, $T(|\mathtt{s}_1|, \ldots, |\mathtt{s}_\mathtt{n}|)$ and $P(|\mathtt{s}_1|, \ldots, |\mathtt{s}_\mathtt{n}|)$ will, respectively, denote the application of $Init$, T and P to this finite instance.

The template of the protocol in Figure 3 is $\mathit{ToyConsensus}(\mathtt{node}, \mathtt{quorum}, \mathtt{value})$. Its finite instance:

$$\begin{aligned} \mathit{ToyConsensus}(3,3,3): \quad & \mathtt{node_3} \triangleq \{\mathtt{n_1}, \mathtt{n_2}, \mathtt{n_3}\} \qquad \mathtt{value_3} \triangleq \{\mathtt{v_1}, \mathtt{v_2}, \mathtt{v_3}\} \\ & \mathtt{quorum_3} \triangleq \{\mathtt{q_{12}} : \{\mathtt{n_1}, \mathtt{n_2}\},\ \mathtt{q_{13}} : \{\mathtt{n_1}, \mathtt{n_3}\},\ \mathtt{q_{23}} : \{\mathtt{n_2}, \mathtt{n_3}\}\} \end{aligned} \tag{5.1}$$

will be used as a running example in the chapter. The finite sorts of this instance are defined as sets of arbitrarily named distinct constants. It should be noted that the constants of the $\mathtt{quorum_3}$ sort are subsets of the $\mathtt{node_3}$ sort that satisfy the non-empty intersection axiom and are named to reflect their symmetric dependence on the $\mathtt{node_3}$ sort. This instance has 9 vote and 3 $\mathit{decision}$ state variables, and a state of this instance corresponds to a complete Boolean assignment to these 12 state variables.

In the sequel, we will use $\hat{\mathcal{P}}$ and $\hat{T}$ as shorthand for $\mathcal{P}(|\mathtt{s_1}|, \ldots, |\mathtt{s_n}|)$ and $T(|\mathtt{s_1}|, \ldots, |\mathtt{s_n}|)$. Quantifier-free formulas will be denoted by lower-case Greek letters (e.g., φ) and quantified formulas by upper-case Greek letters (e.g., Φ). We use primes (e.g., φ') to represent a formula after a single transition step.

5.4 Protocol Symmetries

The symmetry group of $\hat{\mathcal{P}}$ is $G(\hat{\mathcal{P}}) = \bigtimes_{\mathtt{s} \in S} \mathit{Sym}(\mathtt{s})$, where $\mathit{Sym}(\mathtt{s})$ is the symmetric group, i.e., the set of $|\mathtt{s}|!$ permutations of the constants of the set $\mathtt{s}$.[4] In what follows we will use G instead of $G(\hat{\mathcal{P}})$ to reduce clutter. Given a permutation $\gamma \in G$ and an arbitrary protocol relation ρ instantiated with specific sort constants, the *action* of γ on ρ, denoted ρ^γ, is the relation obtained from ρ by permuting the sort constants in ρ according to γ; it is referred to as the γ-*image* of ρ. Permutation $\gamma \in G$ can also act on any formula involving the protocol relations. In particular, the invariance of protocol behavior under permutation of sort constants implies that the action of γ on the (finite) initial state, transition relation, and property formulas causes a syntactic re-arrangement of their sub-formulas while preserving their logical equivalence:

$$\hat{\mathit{Init}}^\gamma \equiv \hat{\mathit{Init}} \qquad \hat{T}^\gamma \equiv \hat{T} \qquad \hat{P}^\gamma \equiv \hat{P} \tag{5.2}$$

Consider next a clause φ which is a disjunction of literals, namely, instantiated protocol relations or their negations. The *orbit* of φ under G, denoted φ^G, is the set of its images φ^γ for all permutations $\gamma \in G$, i.e., $\varphi^G = \{\varphi^\gamma | \gamma \in G\}$. The γ-image of a clause can be viewed as a

[4] We assume familiarity with basic notions from *group theory* including *permutation groups*, *cycle notation*, *group action* on a set, *orbits*, etc., which can be readily found in standard textbooks on Abstract Algebra [114].

syntactic transformation that will either yield a new logically-distinct clause on different literals or simply re-arrange the literals in the clause without changing its logical behavior (by the commutativity and associativity of disjunction). We define the *logical action* of a permutation γ on a clause φ, denoted $\varphi^{L(\gamma)}$, as:

$$\varphi^{L(\gamma)} = \begin{cases} \varphi^{\gamma} & \text{if } \varphi^{\gamma} \not\equiv \varphi \\ \varphi & \text{if } \varphi^{\gamma} \equiv \varphi \end{cases}$$

and the *logical orbit* of φ as $\varphi^{L(G)} = \{\varphi^{L(\gamma)} \big| \, \gamma \in G\}$. With a slight abuse of notation, logical orbit can also be viewed as the conjunction of the logical images:

$$\varphi^{L(G)} = \bigwedge_{\gamma \,\in\, G} \varphi^{L(\gamma)}$$

To illustrate these concepts, consider *ToyConsensus*$(3, 3, 3)$ from (6.1). Its symmetries in cycle notation are as follows:

$$\begin{aligned} \mathit{Sym}(\mathtt{node_3}) \quad &= \{(), (\mathtt{n_1\,n_2}), (\mathtt{n_1\,n_3}), (\mathtt{n_2\,n_3}), (\mathtt{n_1\,n_2\,n_3}), (\mathtt{n_1\,n_3\,n_2})\} \\ \mathit{Sym}(\mathtt{value_3}) \quad &= \{(), (\mathtt{v_1\,v_2}), (\mathtt{v_1\,v_3}), (\mathtt{v_2\,v_3}), (\mathtt{v_1\,v_2\,v_3}), (\mathtt{v_1\,v_3\,v_2})\} \\ G = \; \mathit{Sym}&(\mathtt{node_3}) \times \mathit{Sym}(\mathtt{value_3}) \end{aligned} \tag{5.3}$$

The symmetry group (5.3) of *ToyConsensus*$(3, 3, 3)$ has 36 symmetries corresponding to the 6 $\mathtt{node_3}$ × 6 $\mathtt{value_3}$ permutations. The permutations on $\mathtt{quorum_3}$ are *implicit* and based on the permutations of $\mathtt{node_3}$ since $\mathtt{quorum_3}$ is a dependent sort. Now, consider the example clause:

$$\varphi_1 = \mathit{vote}(\mathtt{n_1}, \mathtt{v_1}) \lor \mathit{vote}(\mathtt{n_1}, \mathtt{v_2}) \lor \mathit{vote}(\mathtt{n_1}, \mathtt{v_3}) \tag{5.4}$$

The orbit of φ_1 consists of 36 syntactically-permuted clauses. However, many of these images are logically equivalent yielding the following logical orbit of just 3 logically-distinct clauses:

$$\begin{aligned} \varphi_1^{L(G)} = [\; &\mathit{vote}(\mathtt{n_1}, \mathtt{v_1}) \lor \mathit{vote}(\mathtt{n_1}, \mathtt{v_2}) \lor \mathit{vote}(\mathtt{n_1}, \mathtt{v_3}) \;] \land \\ [\; &\mathit{vote}(\mathtt{n_2}, \mathtt{v_1}) \lor \mathit{vote}(\mathtt{n_2}, \mathtt{v_2}) \lor \mathit{vote}(\mathtt{n_2}, \mathtt{v_3}) \;] \land \\ [\; &\mathit{vote}(\mathtt{n_3}, \mathtt{v_1}) \lor \mathit{vote}(\mathtt{n_3}, \mathtt{v_2}) \lor \mathit{vote}(\mathtt{n_3}, \mathtt{v_3}) \;] \end{aligned} \tag{5.5}$$

5.5 *SymIC3*: Symmetric Incremental Induction

SymIC3 is an extension of the standard IC3 algorithm [58, 108] that takes advantage of the symmetries in a finite instance $\hat{\mathcal{P}}$ of an unbounded protocol $\mathcal{P}$ to *boost learning* during backward reachability. Specifically, it refines the current frame, in a *single* step, with *all* clauses in the logical orbit $\varphi^{L(G)}$ of a newly learned quantifier-free clause φ. In other words, having determined that the backward 1-step check $F_{i-1} \wedge \hat{T} \wedge [\neg\varphi]'$ is unsatisfiable (i.e., that states in cube $\neg\varphi$ in frame F_i are unreachable from the previous frame F_{i-1}), *SymIC3* refines F_i with $\varphi^{L(G)}$, i.e., $F_i := F_i \wedge \varphi^{L(G)}$, rather than with just φ. Thus, at each refinement step, *SymIC3* not only blocks cube $\neg\varphi$, but also all symmetrically equivalent cubes $[\neg\varphi]^\gamma$ for all $\gamma \in G$. This simple change to the standard incremental induction algorithm significantly improves performance since the extra clauses used to refine F_i a) are derived *without* making additional backward 1-step queries, and b) provide stronger refinement in each step of backward reachability leading to faster convergence with fewer counterexamples-to-induction (CTIs). The proof of correctness of symmetry boosting can be found in Appendix A.2.1.

5.6 Quantifier Inference

The key insight underlying our overall approach is that the explicit logical orbit, in a finite protocol instance, of a learned clause φ can be exactly, and systematically, captured by a corresponding quantified predicate Φ. In retrospect, this should not be surprising since symmetry and quantification can be seen as different ways of expressing invariance under permutation of the sort constants in the clause. To motivate the connection between symmetry and quantification, consider the following quantifier-free clause from our running example and a proposed quantified predicate that *implicitly* represents its logical orbit:

$$\varphi_2 = \neg \mathit{decision}(\mathtt{v}_1) \vee \mathit{decision}(\mathtt{v}_2)$$

$$\Phi_2 = \forall X_1, X_2 \in \mathtt{value}_3 : \text{ (distinct } X_1\ X_2) \rightarrow [\, \neg \mathit{decision}(X_1) \vee \mathit{decision}(X_2)\,] \qquad (5.6)$$

As shown in Table 5.2, the logical orbit $\varphi_2^{L(G)}$ consists of 6 logically-distinct clauses corresponding to the 6 permutations of the 3 constants of the $\mathtt{value}_3$ sort. Evaluating Φ_2 by substituting all $3 \times 3 = 9$ assignments to the variable pair $(X_1, X_2) \in \mathtt{value}_3 \times \mathtt{value}_3$ yields 9 clauses, 3 of which (shown faded) are trivially true since their "distinct" antecedents are false, with the remaining 6 corresponding to each of the clauses obtained through permutations of the 3 $\mathtt{value}_3$ constants. Similarly, we can show that the 3-clause logical orbit $\varphi_1^{L(G)}$ in (5.5) can be succinctly expressed by the quantified predicate:

$$\Phi_1 = \ \forall Y \in \mathtt{node}_3,\ \exists X \in \mathtt{value}_3 : \ \mathit{vote}(Y, X) \qquad (5.7)$$

which employs universal *and* existential quantification. And, finally, φ_3 and Φ_3 below illustrate how a clause whose logical orbit is just itself can also be expressed as an existentially-quantified

(X_1, X_2)	Instantiation of Φ_2	Permutation
$(\texttt{v}_1, \texttt{v}_1)$	(distinct $\texttt{v}_1$ $\texttt{v}_1$) $\rightarrow [\ \neg decision(\texttt{v}_1) \vee decision(\texttt{v}_1)\ ]$	none
$(\texttt{v}_1, \texttt{v}_2)$	(distinct $\texttt{v}_1$ $\texttt{v}_2$) $\rightarrow [\ \neg decision(\texttt{v}_1) \vee decision(\texttt{v}_2)\ ]$	()
$(\texttt{v}_1, \texttt{v}_3)$	(distinct $\texttt{v}_1$ $\texttt{v}_3$) $\rightarrow [\ \neg decision(\texttt{v}_1) \vee decision(\texttt{v}_3)\ ]$	$(\texttt{v}_2\ \texttt{v}_3)$
$(\texttt{v}_2, \texttt{v}_1)$	(distinct $\texttt{v}_2$ $\texttt{v}_1$) $\rightarrow [\ \neg decision(\texttt{v}_2) \vee decision(\texttt{v}_1)\ ]$	$(\texttt{v}_1\ \texttt{v}_2)$
$(\texttt{v}_2, \texttt{v}_2)$	(distinct $\texttt{v}_2$ $\texttt{v}_2$) $\rightarrow [\ \neg decision(\texttt{v}_2) \vee decision(\texttt{v}_2)\ ]$	none
$(\texttt{v}_2, \texttt{v}_3)$	(distinct $\texttt{v}_2$ $\texttt{v}_3$) $\rightarrow [\ \neg decision(\texttt{v}_2) \vee decision(\texttt{v}_3)\ ]$	$(\texttt{v}_1\ \texttt{v}_2\ \texttt{v}_3)$
$(\texttt{v}_3, \texttt{v}_1)$	(distinct $\texttt{v}_3$ $\texttt{v}_1$) $\rightarrow [\ \neg decision(\texttt{v}_3) \vee decision(\texttt{v}_1)\ ]$	$(\texttt{v}_1\ \texttt{v}_3\ \texttt{v}_2)$
$(\texttt{v}_3, \texttt{v}_2)$	(distinct $\texttt{v}_3$ $\texttt{v}_2$) $\rightarrow [\ \neg decision(\texttt{v}_3) \vee decision(\texttt{v}_2)\ ]$	$(\texttt{v}_1\ \texttt{v}_3)$
$(\texttt{v}_3, \texttt{v}_3)$	(distinct $\texttt{v}_3$ $\texttt{v}_3$) $\rightarrow [\ \neg decision(\texttt{v}_3) \vee decision(\texttt{v}_3)\ ]$	none

Table 5.2: Correlation between symmetry and quantification for Φ_2 from (5.6)
Highlighted clauses represent the logical orbit $\varphi_2^{L(G)}$
none indicates the clause has no corresponding permutation $\gamma \in Sym(\texttt{value}_3)$

predicate.

$$\varphi_3 = decision(\texttt{v}_1) \vee decision(\texttt{v}_2) \vee decision(\texttt{v}_3)$$

$$\Phi_3 = \exists\, X \in \texttt{value}_3 :\ decision(X) \tag{5.8}$$

We will first describe basic quantifier inference for protocols with independent sorts. This is done by analyzing the syntactic structure of each quantifier-free clause learned during incremental induction to derive a quantified form that expresses the clause's logical orbit. We later discuss extensions to this approach that consider protocols with dependent sorts, such as $ToyConsensus$, for which the basic single-clause quantifier inference may be insufficient.

5.6.1 Basic Quantifier Inference

Given a quantifier-free clause φ, quantifier inference seeks to derive a *compact* quantified predicate that *implicitly* represents, rather than explicitly enumerates, its logical orbit. The procedure must satisfy the following conditions:

Correctness – The inferred quantified predicate Φ should be logically-equivalent to the explicit logical orbit $\varphi^{L(G)}$.

Compactness – The number of quantified variables in Φ for each sort $\texttt{s} \in S$ should be independent of the sort size $|\texttt{s}|$. Intuitively, this condition ensures that the size of the quantified predicate, measured as the number of its quantifiers, remains bounded for *any* finite protocol instance, and more importantly, for the unbounded protocol.

SymIC3 constructs the orbit's quantified representation by a) inferring the required quantifiers for each sort separately, and b) stitching together the inferred quantifiers for the different sorts to form the final result. The key to capturing the logical orbit and deriving its compact quantified representation is a simple analysis of the *structural distribution* of each sort's constants in the target clause. Let $\pi(\varphi, \mathtt{s})$ be a partition of the constants of sort $\mathtt{s}$ in φ based on whether or not they appear *identically* in the literals of φ. Two constants $\mathtt{c_i}$ and $\mathtt{c_j}$ are identically present in φ if they occur in φ and swapping them results in a logically-equivalent clause, i.e., $\varphi^{(\mathtt{c_i}\ \mathtt{c_j})} \equiv \varphi$. Let $\#(\varphi, \mathtt{s})$ be the number of constants of $\mathtt{s}$ that appear in φ, and let $|\pi(\varphi, \mathtt{s})|$ be the number of classes/cells in $\pi(\varphi, \mathtt{s})$. Consider the following scenarios for quantifier inference on sort $\mathtt{s}$:

5.6.1.A $\#(\varphi, \mathtt{s}) < |\mathtt{s}|$ (infer $\forall$)

In this case, clause φ contains a strict subset of constants from sort $\mathtt{s}$, indicating that the number of literals in φ parameterized by $\mathtt{s}$ constants is *independent* of the sort size $|\mathtt{s}|$. Increasing sort size simply makes the orbit *longer* by adding more symmetrically-equivalent but logically-distinct clauses. An example of this case is φ_2 and Φ_2 in (5.6). The quantified predicate representing such an orbit requires $\#(\varphi, \mathtt{s})$ universally-quantified sort variables corresponding to the $\#(\varphi, \mathtt{s})$ sort constants in the clause, and expresses the orbit as an implication whose antecedent is a "distinct" constraint that ensures that the variables cannot be instantiated with identical constants.

5.6.1.B $\#(\varphi, \mathtt{s}) = |\mathtt{s}|$

When all constants of a sort $\mathtt{s}$ appear in a clause, the above universal quantification yields a predicate with $|\mathtt{s}|$ quantified variables and fails the compactness requirement since the number of quantified variables becomes unbounded as the sort size increases. Correct quantification in this case must be inferred by examining the partition of the sort constants in the clause.

I. **Single-cell Partition i.e.,** $|\pi(\varphi, \mathtt{s})| = 1$ **(infer** $\exists$**)**

When all sort constants appear *identically* in φ, $\pi(\varphi, \mathtt{s})$ is a unit partition. Applying *any* permutation $\gamma \in \mathit{Sym}(\mathtt{s})$ to φ yields a logically-equivalent clause, i.e., the logical orbit in this case is just a single clause. Increasing the size of sort $\mathtt{s}$ simply yields a *wider* clause and suggests that such an orbit can be encoded as a predicate with a single existentially-quantified variable that ranges over all the sort constants. For example, the partition of the $\mathtt{value_3}$ sort constants in φ_1 from (5.4) is $\pi(\varphi_1, \mathtt{value_3}) = \{\{\mathtt{v_1}, \mathtt{v_2}, \mathtt{v_3}\}\}$ since all three constants appear identically in φ_1. The orbit of this clause is just itself and can be encoded as:

$$\Phi_1(\texttt{value}_3) = \ \exists X \in \texttt{value}_3 : \ \mathit{vote}(\texttt{n}_1, X)$$

Also, since $\#(\varphi_1, \texttt{node}_3) < |\texttt{node}_3|$, universal quantification (as in Section 5.6.1.A) correctly captures the dependence of the clause's logical orbit on the $\texttt{node}_3$ sort to get the overall quantified predicate Φ_1 in (5.7).

II. **Multi-cell Partition i.e.,** $|\pi(\varphi, \texttt{s})| > 1$ **(infer** $\forall\exists$**)**

In this case, a fixed number of the constants of sort $\texttt{s}$ appear differently in φ with the remaining constants appearing identically, resulting in a multi-cell partition. Specifically, assume that a number $0 < k < |\texttt{s}|$ exists that is independent of $|\texttt{s}|$ such that $\pi(\varphi, \texttt{s})$ has $k + 1$ cells in which one cell has $|\texttt{s}| - k$ identically-appearing constants and each of the remaining k cells contains one of the differently-appearing constants. It can be shown that the logical orbit in this case can be expressed by a quantified predicate with k universal quantifiers and a single existential quantifier. For example, the partition of the $\texttt{value}_3$ constants in the clause:

$$\varphi_4 = \ \neg \mathit{decision}(\texttt{v}_1) \vee \mathit{decision}(\texttt{v}_2) \vee \mathit{decision}(\texttt{v}_3)$$

is $\pi(\varphi_4, \texttt{value}_3) = \{\{\texttt{v}_1\}, \{\texttt{v}_2, \texttt{v}_3\}\}$ since $\texttt{v}_1$ appears differently from $\texttt{v}_2$ and $\texttt{v}_3$. The logical orbit of this clause is:

$$\begin{aligned}\varphi_4^{L(G)} = & [\,\neg \mathit{decision}(\texttt{v}_1) \vee \mathit{decision}(\texttt{v}_2) \vee \mathit{decision}(\texttt{v}_3)\,] \wedge \\ & [\,\neg \mathit{decision}(\texttt{v}_2) \vee \mathit{decision}(\texttt{v}_1) \vee \mathit{decision}(\texttt{v}_3)\,] \wedge \\ & [\,\neg \mathit{decision}(\texttt{v}_3) \vee \mathit{decision}(\texttt{v}_2) \vee \mathit{decision}(\texttt{v}_1)\,] \end{aligned} \tag{5.9}$$

and can be compactly encoded with an outer universally-quantified variable corresponding to the sort constant in the singleton cell, and an inner existentially-quantified variable corresponding to the other $|\texttt{value}_3| - 1$ identically-present sort constants. A "distinct" constraint must also be conjoined with the literals involving the existentially-quantified variable to exclude the constant corresponding to the universally-quantified variable from the inner quantification. $\varphi_4^{L(G)}$ can thus be shown to be logically-equivalent to:

$$\Phi_4 = \forall Y \in \texttt{value}_3,\ \exists X \in \texttt{value}_3 : \ \neg \mathit{decision}(Y) \vee [(\text{distinct } Y\ X) \wedge \mathit{decision}(X)] \tag{5.10}$$

Combining Quantifier Inference for Different Sorts— The complete quantified predicate Φ representing the logical orbit of clause φ can be obtained by applying the above inference procedure to each sort in φ separately and in any order. This is possible since the sorts are assumed to be independent: the constants of one sort do not permute with the constants of a different sort. This will yield a predicate Φ that has the quantified prenex form $\forall^*\exists^* < \texttt{CNF expression} >$, where all universals for each sort are collected together and precede all the existential quantifiers.

It is interesting to note that this connection between symmetry and quantification suggests that an orbit can be visualized as a two-dimensional object whose height and width correspond, respectively, to the number of universally- and existentially-quantified variables. A proof of the correctness of this quantifier inference procedure can be found in Appendix A.2.2.

5.6.2 Quantifier Inference Beyond $\forall^*\exists^*$

We observed that for some protocols, particularly those that have dependent sorts such as *ToyConsensus*, the above inference procedure violates the compactness requirement. In other words, restricting inference to a $\forall^*\exists^*$ quantifier prefix causes the number of quantifiers to become unbounded as sort sizes increase. Recalling that the $\forall^*\exists^*$ pattern is inferred from the symmetries of a *single* clause, whose literals are the protocol's state variables, suggests that inference of more complex quantification patterns may necessitate that we examine the structural distribution of sort constants across *sets of clauses.* While this is an interesting possible direction for further exploration of the connection between symmetry and quantification, an alternative approach is to take advantage of the *formula structure* of the protocol's transition relation. For example, the transition relation of *ToyConsensus* is specified in terms of two quantified sub-formulas, $didNoteVote$ and $chosenAt$, that can be viewed, in analogy with a sequential hardware circuit, as internal auxiliary non-state variables that act as "combinational" functions of the state variables. By allowing such auxiliary variables to appear explicitly in clauses learned during incremental induction, the quantified predicates representing the logical orbits of these clauses (according to the basic inference procedure in Section 5.6.1) will *implicitly* incorporate the quantifiers used in the auxiliary variable definitions and automatically have a quantifier prefix that generalizes the basic $\forall^*\exists^*$ template.

Revisiting ToyConsensus— When *SymIC3* is run on the finite instance *ToyConsensus*(3,3,3), it terminates with the following two strengthening assertions:

$$A_1 = \forall\, N \in \texttt{node}_3,\, V_1,\, V_2 \in \texttt{value}_3 : \text{(distinct } V_1\ V_2) \rightarrow \neg vote(N, V_1) \vee \neg vote(N, V_2) \tag{5.11}$$

$$A_2 = \forall\, V \in \texttt{value}_3,\, \exists\, Q \in \texttt{quorum}_3 : \neg decision(V) \vee chosenAt(Q, V) \tag{5.12}$$

$$= \forall\, V \in \texttt{value}_3,\, \exists\, Q \in \texttt{quorum}_3 : \neg decision(V) \vee [\, \forall\, N \in Q : vote(N, V)\,]$$

which, together with $\hat{P}$, serve as an inductive invariant proving that $\hat{P}$ holds for this instance. Both assertions are obtained using the basic quantifier inference procedure in Section 5.6.1 that produces a $\forall^*\exists^*$ quantifier prefix in terms of the clause variables. Note, however, that A_2 is expressed in terms of the auxiliary variable $chosenAt$. Substituting the definition of $chosenAt$ yields an assertion with a $\forall\exists\forall$ quantifier prefix exclusively in terms of the protocol's state variables.

5.7 Finite Convergence Checks

Given a safe finite instance $\hat{\mathcal{P}} \triangleq \mathcal{P}(|\mathtt{s_1}|, \ldots, |\mathtt{s_n}|)$, let $Inv_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}$ denote the inductive invariant derived by *SymIC3* to prove that $\hat{P}$ holds in $\hat{\mathcal{P}}$. What remains is to determine the instance size $|\mathtt{s_1}|, \ldots, |\mathtt{s_n}|$ needed so that $Inv_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}$ is also an inductive invariant for all sizes. If the instance size is too small, $\hat{\mathcal{P}}$ may not include all protocol behaviors and $Inv_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}$ will not be inductive at larger sizes. As shown in the invisible invariant approach [215, 36, 237, 39, 202], increasing the instance size becomes necessary to include new protocol behaviors missing in $\hat{\mathcal{P}}$, until protocol behaviors *saturate*. We propose an *automatic* way to update the instance size and reach saturation by starting with an initial *base size* and iteratively increasing the size until *finite convergence* is achieved.

The initial base size can be chosen to be any non-trivial instance size and can be easily determined by a simple analysis of the protocol description. For example, any non-trivial instance of the *ToyConsensus* protocol should have $|\mathtt{node}| \geq 3$, $|\mathtt{quorum}| \geq 3$, and $|\mathtt{value}| \geq 2$.

Our finite convergence procedure can be seen as an integration of symmetry saturation and a stripped-down form of multi-dimensional mathematical induction and has similarities with previous works on structural induction [160, 117] and proof convergence [104]. To determine if $Inv_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}$ is inductive for any size, the procedure performs the following checks for $1 \leq i \leq \mathtt{n}$:

$$\text{a) } Init(|\mathtt{s_1}|..|\mathtt{s_i}|+1..|\mathtt{s_n}|) \rightarrow Inv_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}(|\mathtt{s_1}|..|\mathtt{s_i}|+1..|\mathtt{s_n}|) \tag{5.13}$$

$$\text{b) } Inv_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}(|\mathtt{s_1}|..|\mathtt{s_i}|+1..|\mathtt{s_n}|) \wedge T(|\mathtt{s_1}|..|\mathtt{s_i}|+1..|\mathtt{s_n}|) \rightarrow Inv'_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}(|\mathtt{s_1}|..|\mathtt{s_i}|+1..|\mathtt{s_n}|) \tag{5.14}$$

where $Inv_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}(|\mathtt{s_1}|..|\mathtt{s_i}|+1..|\mathtt{s_n}|)$ denotes the application of $Inv_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}$ to an instance in which the size of sort $\mathtt{s_i}$ is increased by 1 while the sizes of the other sorts are unchanged.[5]

If all of these checks pass, we can conclude that $Inv_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}$ is not specific to the instance size used to derive it and that we have reached *cutoff*, i.e., that $Inv_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}$ is an inductive invariant for *any* size. Intuitively, this suggests that adding a new protocol component (e.g., client, server, node, proposer, acceptor) does not add any unseen unique behavior, and hence proving safety till the cutoff is sufficient to prove safety for any instance size.

[5] Sort dependencies, if any, should be considered when increasing a sort size.

On the other hand, failure of these checks, say for sort $\mathtt{s_i}$, implies that $\mathit{Inv}_{|\mathtt{s_1}|\ldots|\mathtt{s_n}|}$ will fail for larger sizes and cannot be inductive in the unbounded case, and we need to repeat *SymIC3* on a finite instance with an increased size for sort $\mathtt{s_i}$, i.e., $\hat{\mathcal{P}}_{new} \triangleq \mathcal{P}(|\mathtt{s_1}|, .., |\mathtt{s_i}| + \mathtt{1}, .., |\mathtt{s_n}|)$, to include new protocol behaviors that are missing in $\hat{\mathcal{P}}$.

Recall from (5.11) and (5.12), running *SymIC3* on $\mathit{ToyConsensus}(3, 3, 3)$ produces $\mathit{Inv}_{3,3,3} = A_1 \wedge A_2 \wedge \hat{P}$. $\mathit{Inv}_{3,3,3}$ passes checks (5.13) and (5.14) for instances $\mathit{ToyConsensus}(4, 4, 3)$ and $\mathit{ToyConsensus}(3, 3, 4)$, indicating finite convergence.[6] $\mathit{Inv}_{3,3,3}$ passes standard induction checks in the unbounded domain as well, establishing it as a proof certificate that proves the property as safe in $\mathit{ToyConsensus}$.

5.8 Simple Enhancements

We also explored few simple enhancements to symmetric incremental induction that strengthen the inferred quantified predicates whenever safely possible to do during incremental induction by a) dropping the “distinct” antecedent, and b) rearranging the quantifiers if the strengthened predicate is still unreachable from the previous frame.

5.8.1 Antecedent Reduction

Antecedent reduction strengthens a quantified predicate Φ by dropping the antecedent (distinct $\ldots$) and checking the unsatisfiability of the query [$F_{i-1} \wedge \hat{T} \wedge \neg\Phi'$]. For example, Φ_2 from (5.6) can possibly be strengthened by dropping (distinct X_1 X_2) from the antecedent to get Φ_{new}, if the query [$F_{i-1} \wedge \hat{T} \wedge \neg\Phi'_{new}$] is unsatisfiable, where

$$\Phi_{new} = \ \forall X_1, X_2 \in \mathtt{value} : \ \neg \mathit{decision}(X_1) \vee \mathit{decision}(X_2)$$

If instead, the query is satisfiable, the original predicate is learnt.

5.8.2 Pushing out Existential Quantifiers

We can additionally strengthen the quantified predicate Φ by *pushing out* existential quantifiers whenever safe to do so, by checking the unsatisfiability of the query [$F_{i-1} \wedge \hat{T} \wedge \neg\Phi'$]. For example, Φ_4 from (5.10) can possibly be strengthened by pushing out $\exists X$ to get Φ_{new}, if the query [$F_{i-1} \wedge \hat{T} \wedge \neg\Phi'_{new}$] is unsatisfiable, where

$$\Phi_{new} = \exists X \in \mathtt{value}_3,\ \forall Y \in \mathtt{value}_3 : \ \neg \mathit{decision}(Y) \vee [(\text{distinct } Y\ X) \wedge \mathit{decision}(X)]$$

[6] Since quorum is a dependent sort on node, it is increased together with the node sort.

If instead the query is unsatisfiable, we learn the original form. Logically, *pushing out* the existential quantifier results in a reduced/stricter formula, with $\Phi_{new} \rightarrow \Phi$, but $\Phi \not\rightarrow \Phi_{new}$.

5.9 IC3PO: IC3 for Proving Protocol Properties

Given a protocol specification $\mathcal{P}$, IC3PO iteratively invokes *SymIC3* on finite instances of increasing size, starting with a given initial base size. Upon termination, IC3PO either a) reaches convergence on an inductive invariant $Inv_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}$ that proves P for the unbounded protocol $\mathcal{P}$, or b) produces a counterexample trace $Cex_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}$ that serves as a finite witness to its violation in both the finite instance and the unbounded protocol. The detailed pseudo code of IC3PO is available in Appendix A.1.

Our implementation of IC3PO is publicly available at `https://github.com/aman-goel/ic3po`. The implementation accepts protocol descriptions in the Ivy language [210] and uses the Ivy compiler to extract a quantified, logical formulation $\mathcal{P}$ in a customized VMT [26] format. We use a modified version [12] of the pySMT [115] library to implement our prototype and have an option to use Z3 [98] or Yices 2 [107] SMT solver for solver reasoning. We use the SMT-LIB [46] theory of free sorts and function symbols with datatypes and quantifiers (`UFDT`), which allows formulating SMT queries for both, the finite and the unbounded domains. By default, IC3PO uses quantifier-free SMT solving using Yices 2 solver, where quantifier elimination is performed for each finite-domain SMT query by expanding the universal and existential quantifiers over the corresponding finite domain as AND and OR respectively.

For a safe protocol, the inductive proof is printed in the Ivy format as an *independently checkable* proof certificate, which can be further validated with the Ivy verifier.

5.10 Evaluation

We evaluated IC3PO on a total of 29 distributed protocols including 4 problems from [186], 13 from [154], and 12 from [14]. This evaluation set includes fairly complex models of consensus algorithms as well as protocols such as two-phase commit, chord ring, hybrid reliable broadcast, etc. Several studies [138, 210, 186, 154, 113, 49] have indicated the challenges involved in verifying these protocols.

All 29 protocols are safe based on manual verification. Even though finding counterexample traces is equally important, we limit our evaluation to safe protocols where the property holds, since inferring inductive invariants is the main bottleneck of existing techniques for verifying distributed protocols [112, 210, 111].

We compared IC3PO against the following 3 verifiers that implement state-of-the-art IC3-style techniques for automatic verification of distributed protocols:

- SWISS [136] uses SMT solving to derive an inductive invariant by performing an enumerative search in an optimized and bounded invariant search space.
- fol-ic3 [154] is a recent technique implemented in *mypyvy* that extends IC3 with the ability to infer inductive invariants with quantifier alternations.
- DistAI [235] performs data-driven invariant learning by enumerating over possible invariants derived from simulating a protocol at different instance sizes, followed by iteratively refining and checking candidate invariants.
- I4 [186, 187] performs finite-domain IC3 (without accounting for symmetry) using the AVR model checker [124, 125], followed by iteratively generalizing and checking the inductive invariant produced by AVR using Ivy.
- UPDR is the implementation of the $\text{PDR}^{\forall}$/UPDR algorithm [151] for verifying distributed protocols, from the *mypyvy* [16] framework.

All experiments were performed on an Intel (R) Xeon CPU (X5670). For each run, we used a timeout of 1 hour and a memory limit of 32 GB. All tools were executed in their respective default configurations. We used Z3 [98] version 4.8.9, Yices 2 [107] version 2.6.2, and CVC4 [45] version 1.7.

5.10.1 Aggregate Results

Tables 5.3 & 5.1 summarize the experimental results. Apart from the number of problems solved, we compared the tools on 3 metrics: run time in seconds, proof size measured by the number of assertions in the inductive invariant for the unbounded protocol, and the total number of SMT queries made. Each tool uses SMT queries differently (e.g., I4 uses `QF_UF` for finite, `UF` for unbounded). Comparing the number of SMT queries still helps in understanding the run time behavior.

IC3PO solved all 29 problems, SWISS solved 22, fol-ic3 solved 23, DistAI solved 17, I4 solved 13, and UPDR solved 14 problems. Overall, compared to the other tools IC3PO is faster, requires fewer SMT queries, and produces shorter inductive proofs even for problems requiring inductive invariants with quantifier alternations (marked with Æ in Table 5.3).

	Human		IC3PO			SWISS			fol-ic3			DistAI	
Protocol (#29)	Inv	info	Time	Inv	SMT	Time	Inv	SMT	Time	Inv	SMT	Time	Inv
tla-consensus	1		**0**	1	17	1	1	2	1	1	29	1	1
tla-tcommit	3		**1**	2	31	4	9	201	2	3	162	1	7
i4-lock-server	2		**1**	2	37	4	1	255	1	2	66	2	2
ex-quorum-leader-election	3		**3**	5	129	20	4	1616	24	8	1078	26	11
pyv-toy-consensus-forall	4		**3**	4	105	4	4	252	11	5	587	22	8
tla-simple	8		**6**	3	285	59	8	2576	timeout			291	485
ex-lockserv-automaton	2		7	12	594	timeout			10	12	1181	**2**	13
tla-simpleregular	9		**8**	4	346	196	18	6030	57	9	314	188	314
pyv-sharded-kv	5		10	8	590	timeout			22	10	522	**2**	16
pyv-lockserv	9		11	12	702	timeout			8	11	1044	**2**	13
tla-twophase	12		14	10	984	43	24	3147	9	12	1635	**2**	61
i4-learning-switch	8		**14**	9	589	timeout			timeout			50	32
ex-simple-decentralized-lock	5		19	15	2219	12	3	953	**4**	8	291	18	17
i4-two-phase-commit	11		27	11	2541	21	17	1444	8	9	1083	**4**	31
pyv-consensus-wo-decide	5		**50**	9	1886	84	6	5963	168	26	5692	96	44
pyv-consensus-forall	7		**99**	10	3445	148	8	10124	2461	27	16182	1122	58
pyv-learning-switch	8		**127**	13	3388	3460	8	31091	timeout			unknown	
i4-chord-ring-maintenance	18		229	12	6418	timeout			timeout			**120**	164
pyv-sharded-kv-no-lost-keys	2	Æ	**3**	2	57	14	3	856	3	2	51	unknown	
ex-naive-consensus	4	Æ	**6**	4	239	37	4	2606	73	18	414	unknown	
pyv-client-server-ae	2	Æ ≜	**2**	2	49	33	4	2763	877	15	700	unknown	
ex-simple-election	3	Æ ≜	7	4	268	**3**	3	163	32	10	222	unknown	
pyv-toy-consensus-epr	4	Æ ≜	**9**	4	370	27	4	1933	70	14	217	unknown	
ex-toy-consensus	3	Æ ≜	10	3	209	**2**	3	123	21	8	124	unknown	
pyv-client-server-db-ae	5	Æ ≜	**17**	6	868	257	12	18266	timeout			unknown	
pyv-hybrid-reliable-broadcast	8	Æ ≜	**587**	4	1474	timeout			1360	23	3387	unknown	
pyv-firewall	2	Æ ⇆	**2**	3	131	unknown		489	7	8	116	unknown	
ex-majorityset-leader-election	5	Æ ⇆	72	7	1552	**50**	9	3486	timeout			error	
pyv-consensus-epr	7	Æ ≜ ⇆	1300	9	29601	**108**	7	8015	1468	30	3355	timeout	
No. of problems solved (out of 29)			**29**			22			23			17	

Table 5.3: Comparison of IC3PO against other state-of-the-art verifiers
Time: run time (seconds), Inv: # assertions in inductive proof, SMT: # SMT queries,
Column "info" provides information on the strengthening assertions (i.e., A) in IC3PO's inductive proof: Æ indicates A has quantifier alternations, ≜ means A has definitions, and ⇆ means A adds quantifier-alternation cycles

5.10.2 Effect of Symmetry Boosting in Incremental Induction

Table 5.4 compares the effects of symmetry boosting in incremental induction for the problems solved by both IC3PO and I4. The table compares the number of SMT solver calls made and counterexamples-to-induction (CTI) encountered during the incremental induction procedure, as well as the number of assertions in the final quantified inductive invariant. *SymIC3*'s symmetry boosting helps IC3PO to make orders of magnitude fewer SMT solver calls compared to I4 and solve the problem after discovering many fewer CTIs. These results justify the runtime speedups observed in Tables 5.3 & 5.1 and confirm the benefits of symmetry-aware learning.

	IC3PO			I4		
Protocol (#13)	SMT	CTI	Inv	SMT	CTI	Inv
tla-consensus	13	0	1	7	0	1
i4-lock-server	31	1	2	35	2	2
ex-quorum-leader-election	117	7	5	15429	847	14
tla-simple	273	23	3	1319	41	3
ex-lockserv-automaton	568	51	12	1731	156	15
pyv-sharded-kv	572	25	8	2101	170	15
pyv-lockserv	676	58	12	1606	142	15
i4-learning-switch	567	32	9	26345	1310	11
ex-simple-decentralized-lock	2155	87	15	5561	490	22
i4-two-phase-commit	2131	68	11	4045	288	16
pyv-consensus-wo-decide	1866	141	9	41137	2451	42
pyv-consensus-forall	3423	247	10	156838	10316	44
pyv-learning-switch	3352	112	13	51021	3639	49
$\sum$ #SMT	15744	(**19.5x** better)		307175		
$\sum$ #CTI	852	(**23.3x** better)		19852		
$\sum$ #Inv	110	(**2.3x** better)		249		

Table 5.4: Comparison of different incremental induction metrics between IC3PO and I4 for problems solved by both
SMT: number of solver queries, CTI: number of counterexamples-to-induction
Inv: number of assertions in the final quantified inductive invariant

5.10.3 Comparison against Human-Written Invariants

Figure 5.1 compares IC3PO's automatically-generated inductive invariants against the human-written proofs on several metrics– the number of assertions, the number of literals, and the number of universal/existential quantifiers. Our evaluation shows IC3PO produces compact proofs of sizes comparable to the manually-written inductive invariants, even shorter than the human proofs on several occasions. As a side benefit, IC3PO's inductive invariants are pretty printed in the Ivy format [13], and thus, can also be independently checked/validated through Ivy.

5.10.4 Discussion

DistAI, I4, and UPDR are limited to generating only universally-quantified invariants over state variables, and hence, were unable to solve any problem that required inferring invariants with a combination of universal and existential quantifiers. Apart from IC3PO, SWISS and fol-ic3 were the only techniques with the capability to infer invariants with quantifier alternations.

Comparing IC3PO and I4 clearly shows the benefits of symmetric incremental induction. For example, I4 requires 7814 SMT queries to eliminate 443 CTIs when solving *ToyConsensus*(3,3,3),

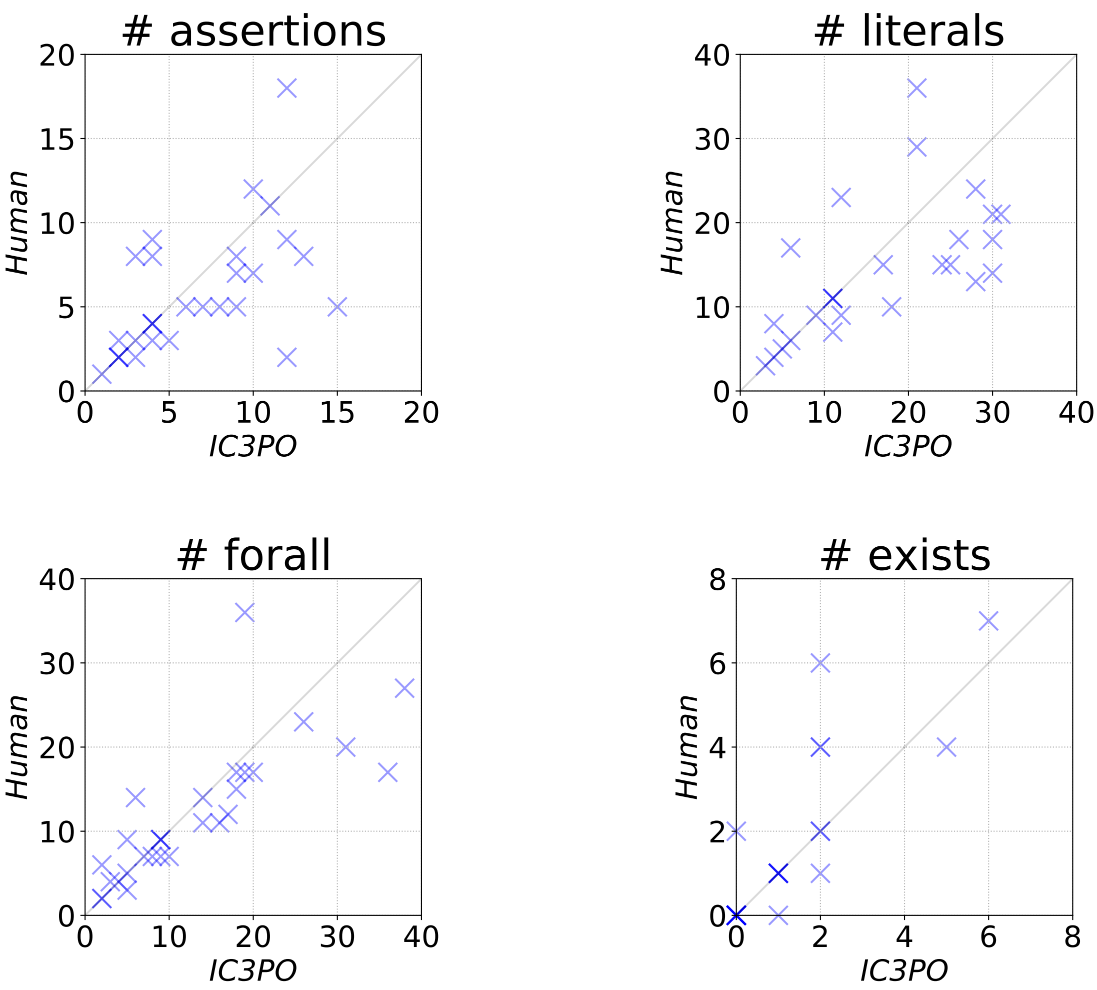


Figure 5.1: Comparison of IC3PO's inductive invariant against *human-written* proof
IC3PO is on x-axis, *human-written* on y-axis

compared to 192 SMT calls and 13 CTIs for IC3PO. Even though both techniques perform finite incremental induction, symmetry boosting in IC3PO leads to a factorial reduction in the number of SMT queries and yields compact inductive proofs.

Comparing IC3PO and UPDR reveals the benefits of finite-domain reasoning methods compared to direct unbounded verification. Even in cases where existential quantifier inference isn't necessary, symmetry-aware finite-domain reasoning gives IC3PO a clear advantage, both in terms of run time and the number of SMT queries.

Comparing IC3PO and DistAI shows the advantage of IC3PO's incremental induction-based approach compared to the data-driven approach of DistAI. Even though DistAI's simulation-based approach showed limited runtime advantage in a few cases, the inductive invariants produced by DistAI were generally too verbose compared to the much concise invariants generated by IC3PO. For example, for the Chord ring protocol [223], DistAI produced an invariant with 166 assertions compared to IC3PO's inductive proof composed of just 12 assertions.

Comparing IC3PO and fol-ic3 highlights the advantage of IC3PO's approach over the separators-based technique [154] used in fol-ic3. The significant performance edge that IC3PO has over fol-ic3 is since a) reasoning in IC3PO is primarily in a (small) finite domain compared to fol-ic3's unbounded reasoning, and b) unlike fol-ic3 which enumeratively searches for specific quantifier patterns, IC3PO finds the required invariants without search by automatically inferring their patterns through an algorithmic syntactic analysis of the symmetry orbit by correlating symmetry with quantification.

Comparing IC3PO and SWISS suggests the complementary advantages offered by IC3PO over the enumeration-based inference technique of SWISS. As the invariant search space increases, the number of SMT queries required by SWISS grows drastically compared to the symmetry-aware property-directed reachability performed by IC3PO.

Overall, the evaluation confirms our main hypothesis, that it is possible to use the relationship between symmetry and quantification to scale the verification of distributed protocols beyond the current state-of-the-art.

5.11 Case Study: Multi-Signature Smart Contract

In a preliminary attempt to explore the applicability of IC3PO in blockchain applications, we applied IC3PO to automatically verify safety of a simple example of a multi-signature smart contract [71, 72] from the Cardano blockchain [9]. Given amount $value$ of some cryptocurrency in a n-of-m multi-signature contract, an approval of at least n out of an *a priori* fixed set of $m \geq n$ owners is required to spend $value$. To collect these n signatures on-chain and without any out-of-band communication, the authors of [71, 72] propose using a state machine operating according to the transition diagram in Figure 5.2. We assume that the threshold n equals a majority of authorized signatures $sigs\_auth$ with $|sigs\_auth| = m$ baked into the contract code.

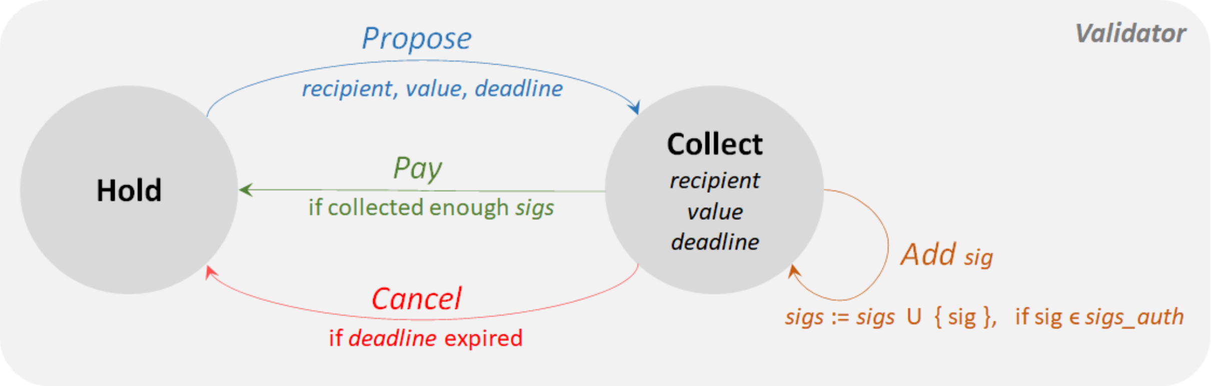


Figure 5.2: A simple multi-signature smart contract

Algorithm 4 presents the multi-signature smart contract modeled in the TLA+ language. The contract consists of 6 sorts (line 1) and 6 state relations defined on these sorts (lines 2-3) that encode the state of the transition system. There are 5 possible actions that can occur, one each corresponding to proposing a transaction (line 6), adding a valid signature (line 7), approving the payment if

Algorithm 4 Multi-Signature smart contract in the TLA+ language

```
1 CONSTANTS $\mathtt{validator}, \mathtt{destination}, \mathtt{value}, \mathtt{deadline}, \mathtt{signature}, \mathtt{quorum}$
2 VARIABLES $holding, collect, sig, sig\_auth, proposed, paid, canceled, expired$
3 $holding \in \mathtt{validator} \rightarrow \text{BOOLEAN}$
  $collect \in (\mathtt{validator} \times \mathtt{destination} \times \mathtt{value} \times \mathtt{deadline}) \rightarrow \text{BOOLEAN}$
  $sig \in (\mathtt{validator} \times \mathtt{destination} \times \mathtt{value} \times \mathtt{deadline} \times \mathtt{signature}) \rightarrow \text{BOOLEAN}$
  $sig\_auth \in \mathtt{signature} \rightarrow \text{BOOLEAN}$
  $proposed \in (\mathtt{validator} \times \mathtt{destination} \times \mathtt{value} \times \mathtt{deadline}) \rightarrow \text{BOOLEAN}$
  $paid \in (\mathtt{validator} \times \mathtt{destination} \times \mathtt{value} \times \mathtt{deadline}) \rightarrow \text{BOOLEAN}$
  $canceled \in (\mathtt{validator} \times \mathtt{destination} \times \mathtt{value} \times \mathtt{deadline}) \rightarrow \text{BOOLEAN}$
  $expired \in \mathtt{deadline} \rightarrow \text{BOOLEAN}$
4 ASSUME $\forall\, Q \in \mathtt{quorum} : Q \subseteq \mathtt{signature} \;\wedge\; \forall\, Q_1,\, Q_2 \in \mathtt{quorum} : Q_1 \cap Q_2 \neq \{\}$
5 $authorized(n,\, k,\, v,\, d) \;\triangleq\; \exists\, Q \in \mathtt{quorum} : \forall\, S \in Q : sig(n,\, k,\, v,\, d,\, S)$
6 $Propose(n, k, v, d) \;\triangleq$
    $\wedge\; holding(n) \;\wedge\; \neg collect(n, k, v, d) \;\wedge\; \neg paid(n, k, v, d) \;\wedge\; \neg canceled(n, k, v, d)$
    $\wedge\; holding' = [holding$ EXCEPT $![n] = \bot]$
    $\wedge\; collect' = [collect$ EXCEPT $![n, k, v, d] = \top]$
    $\wedge\; sig' = [sig$ EXCEPT $![n, k, v, d] = \bot]$
    $\wedge\; proposed' = [proposed$ EXCEPT $![n, k, v, d] = \top]$
    $\wedge$ UNCHANGED $sig\_auth, paid, canceled, expired$
7 $AddSig(n, k, v, d, s) \;\triangleq$
    $\wedge\; collect(n, k, v, d) \;\wedge\; \neg expired(d) \;\wedge\; sig\_auth(s)$
    $\wedge\; sig' = [sig$ EXCEPT $![n, k, v, d, s] = \top]$
    $\wedge$ UNCHANGED $holding, collect, sig\_auth, proposed, paid, canceled, expired$
8 $Pay(n, k, v, d) \;\triangleq$
    $\wedge\; collect(n, k, v, d) \;\wedge\; \neg expired(d) \;\wedge\; authorized(n, k, v, d)$
    $\wedge\; paid' = [paid$ EXCEPT $![n, k, v, d] = \top]$
    $\wedge\; holding' = [holding$ EXCEPT $![n] = \top]$
    $\wedge\; collect' = [collect$ EXCEPT $![n, k, v, d] = \bot]$
    $\wedge$ UNCHANGED $sig, sig\_auth, proposed, canceled, expired$
9 $Expire(d) \;\triangleq$
    $\wedge\; expired' = [expired$ EXCEPT $![d] = \top]$
    $\wedge$ UNCHANGED $holding, collect, sig, sig\_auth, proposed, paid, canceled$
```

(continued)

```
10 $Cancel(n, k, v, d) \triangleq$
     $\land\ collect(n, k, v, d) \land expired(d)$
     $\land\ canceled' = [canceled\ \text{EXCEPT}\ ![n, k, v, d] = \top]$
     $\land\ holding' = [holding\ \text{EXCEPT}\ ![n] = \top]$
     $\land\ collect' = [collect\ \text{EXCEPT}\ ![n, k, v, d] = \bot]$
     $\land\ \text{UNCHANGED}\ sig, sig\_auth, proposed, paid, expired$
11 $Init \triangleq \forall N \in \texttt{validator}:$
                 $\land\ holding(N)$
                 $\land\ \forall K \in \texttt{destination}, V \in \texttt{value}, D \in \texttt{deadline}:$
                     $\land\ \neg collect(N, K, V, D)$
                     $\land\ \neg proposed(N, K, V, D)$
                     $\land\ \neg paid(N, K, V, D)$
                     $\land\ \neg canceled(N, K, V, D)$
                     $\land\ \forall S \in \texttt{signature}: \neg sig(N, K, V, D, S)$
12 $T \triangleq \exists N \in \texttt{validator}, K \in \texttt{destination}, V \in \texttt{value}, D \in \texttt{deadline}:$
                     $\lor\ Propose(N, K, V, D)$
                     $\lor\ \exists S \in \texttt{signature}: CollectSig(N, K, V, D, S)$
                     $\lor\ Pay(N, K, V, D)$
                     $\lor\ Expire(D)$
                     $\lor\ Cancel(N, K, V, D)$
13 $P_1 \triangleq \forall N \in \texttt{validator}, K \in \texttt{destination}, V \in \texttt{value}, D \in \texttt{deadline}:$
       $cancel(N, K, V, D) \rightarrow expired(D)$
14 $P_2 \triangleq \forall N \in \texttt{validator}, K \in \texttt{destination}, V \in \texttt{value}, D \in \texttt{deadline}:$
       $paid(N, K, V, D) \rightarrow (\exists Q \in \texttt{quorum}: \forall S \in Q: sig(N, K, V, D, S) \land sig\_auth(S))$
15 $P_3 \triangleq \forall N \in \texttt{validator}, K \in \texttt{destination}, V \in \texttt{value}, D \in \texttt{deadline}:$
       $paid(N, K, V, D) \rightarrow proposed(N, K, V, D)$
16 $P \triangleq P_1 \land P_2 \land P_3$
```

enough signatures are authorized within the deadline (line 8), expiration of the deadline (line 9), and canceling the transaction if the deadline has expired (line 10). The correctness of the contract requires satisfying 3 safety conditions (lines 13-15) which express the following facts:

P_1: A validator can only cancel a transaction after the deadline has expired.

P_2: A validator can make a payment only if it collects enough valid signatures.

P_3: If a payment is made by a validator, then the recipient is the one proposed in the transaction.

Starting with an initial size of $|\texttt{validator}| = 2$, $|\texttt{destination}| = 2$, $|\texttt{value}| = 2$, $|\texttt{deadline}| = 2$, $|\texttt{signature}| = 3$, and $|\texttt{quorum}| = 3$, in less than 15 seconds, IC3PO proved the correctness of this smart contract by automatically deriving the inductive invariant

$Inv \triangleq P_1 \wedge P_2 \wedge P_3 \wedge A_1 \wedge A_2$ where

$$\begin{aligned} A_1 = {} & \forall N \in \texttt{validator}, K \in \texttt{destination}, V \in \texttt{value}, D \in \texttt{deadline} : \\ & \qquad \mathit{collect}(N, K, V, D) \rightarrow \mathit{proposed}(N, K, V, D) \\ A_2 = {} & \forall N \in \texttt{validator}, K \in \texttt{destination}, V \in \texttt{value}, D \in \texttt{deadline}, S \in \texttt{signature} : \\ & \qquad \mathit{sig}(N, K, V, D, S) \rightarrow \mathit{sig\_auth}(S) \end{aligned}$$

are two additional automatically-generated strengthening assertions that express the following facts about the contract:

A_1: A validator can collect signatures only after the transaction is proposed.

A_2: All signatures collected by the validator are valid.

This preliminary exploration provides encouraging evidence to further explore the applicability of our proposed techniques to verify smart contracts and consensus protocols used in blockchain applications. We leave this exploration as future work.

5.12 Summary

IC3PO is, to our knowledge, the first verification system that uses the synergistic relationship between symmetry and quantification to automatically infer the quantified inductive invariants required to prove the safety of symmetric protocols. Recognizing that symmetry and quantification are alternative ways of capturing invariance, IC3PO extends the incremental induction algorithm to learn clause orbits and encodes these orbits with corresponding logically-equivalent and compact quantified predicates. IC3PO employs a systematic procedure to check for finite convergence, and outputs quantified inductive invariants, with both universal and existential quantifiers, that hold for all protocol parameters. Our evaluation demonstrates that IC3PO is a significant improvement over the current state-of-the-art.

CHAPTER 6

Towards an Automatic Proof of Lamport's Paxos

In this chapter, we continue the description of our proposed approach, implemented in the IC3PO protocol verifier, to *automatically infer the required inductive invariant* for an unbounded distributed protocol by adding simple extensions to the finite-domain incremental induction algorithm. *Symmetry boosting*, introduced in the last chapter, takes advantage of a protocol's *spatial* regularity to automatically infer quantified strengthening assertions that reflect the protocol's structural symmetries. This chapter describes two additional techniques: *range boosting* and *hierarchical strengthening* which take advantage, respectively, of a protocol's *temporal* regularity and hierarchical structure, and demonstrates how IC3PO was used to automatically obtain an inductive invariant for Lamport's Paxos protocol [165, 167].

Lamport's celebrated Paxos consensus protocol forms the basis for implementing many efficient and highly fault-tolerant distributed services [66, 74, 148]. Paxos is generally viewed as a complex hard-to-understand algorithm. Notwithstanding its complexity, in this chapter, we take a step towards automatically proving the safety of Paxos by taking advantage of three structural features in its specification: *spatial regularity* in its unordered domains, *temporal regularity* in its totally-ordered domain, and its *hierarchical composition*. By carefully integrating these structural features in IC3PO, we were able to infer an inductive invariant that identically matches the human-written one previously derived with significant manual effort using interactive theorem proving. While various attempts have been made to verify different versions of Paxos, to the best of our knowledge, this is the first demonstration of an automatically-inferred inductive invariant for Lamport's original Paxos specification. We note that these structural features are not specific to Paxos and that IC3PO can serve as an automatic general-purpose protocol verification tool.

The chapter is structured as follows: §6.1 presents preliminaries. §6.2 and §6.3 describe range boosting and hierarchical strengthening. §6.4 details the four-level hierarchy we used to prove Paxos and §6.5 is a record of the IC3PO run showing the actual assertions it inferred at each level of the hierarchy. §6.6 discusses some of the features and interesting details on this automatically-generated proof. Experimental comparisons with other approaches are provided in §6.7 and the chapter concludes with a summary in §6.8.

6.1 Preliminaries

6.1.1 Notation

We will use Init, T, and P to denote the quantified formulas that specify, respectively, a protocol's initial states, its transition relation, and the safety property that is required to hold on all reachable states. We use primes (e.g., φ') to represent a formula after a single transition step. The notation $V!A$ (resp. ϵA, $I!A$, and $P!A$) means that assertion A was inferred by IC3PO for the *Voting* (resp. *SimplePaxos*, *ImplicitPaxos*, and *Paxos*) protocol.

As an example, consider a protocol $\mathcal{P}$ with two sorts, a symmetric sort $\mathtt{aSort}$ and a totally-ordered sort $\mathtt{bSort}$, along with relations $p(\mathtt{aSort}, \mathtt{bSort})$ and $q(\mathtt{bSort})$ defined on these sorts. Viewed as a parameterized system $\mathcal{P}(\mathtt{aSort}, \mathtt{bSort})$, we can specify its finite instance $\mathcal{P}(3, 4)$ as:

$$\begin{aligned} \mathcal{P}(3,4): \qquad & \mathtt{aSort}_3 \triangleq \{\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3\} \\ & \mathtt{bSort}_4 \triangleq [\mathtt{b}_{\mathtt{min}}, \mathtt{b}_1, \mathtt{b}_2, \mathtt{b}_{\mathtt{max}}] \end{aligned} \tag{6.1}$$

where $\mathtt{aSort}_3$ represents the finite symmetric sort of this instance defined as a set of arbitrarily-named distinct constants, while the finite totally-ordered sort $\mathtt{bSort}_4$ is composed of a list of ordered constants, i.e., $\mathtt{b}_{\mathtt{min}} < \mathtt{b}_1 < \mathtt{b}_2 < \mathtt{b}_{\mathtt{max}}$. This instance can be encoded using twelve p and four q BOOLEAN state variables. A *state* of this instance corresponds to a complete assignment to these 16 state variables, with a total state-space size of 2^{16}. We will use $\widehat{T}$ instead of T to denote the transition relation of the finite instance.

6.1.2 Clause Boosting and Quantifier Inference

As described in the previous chapter, our basic framework for inferring the quantified assertions required to prove protocol safety extends the finite IC3/PDR incremental induction algorithm by *boosting* its clause learning during the 1-step backward reachability checks. Specifically, a clause φ is learned in (and refines) frame F_i if the 1-step query $\psi_i := F_{i-1} \wedge \widehat{T} \wedge [\neg\varphi']$ is unsatisfiable. This means that cube $\neg\varphi$ in frame F_i is unreachable from frame F_{i-1}. Boosting refers to: a) "growing" φ to a set of clauses that also satisfy this *unreachability constraint* from frame F_{i-1}, and b) refining the frame F_i with the entire clause set instead of just φ. Such boosting accelerates the convergence of incremental induction but, more importantly, makes it possible, under some regularity assumptions, to represent this set of clauses by a *single logically-equivalent quantified clause* Φ and is the key to generalizing the results of such finite analysis to unbounded domains.

For a symmetric sort, the constants in the sort represent a finite set of k identical processes that are essentially indistinguishable *replicas* which can be permuted arbitrarily without changing the protocol behavior. A learned clause φ parameterized by the constants of such a sort can be boosted by permuting its constants in all possible $k!$ ways yielding a set of symmetrically-equivalent clauses, i.e., its symmetry *orbit* φ^{Sym_k} under the full symmetric group Sym_k. By construction, all clauses in φ's orbit automatically satisfy the unreachability constraint without the need to perform additional 1-step queries. Furthermore, as shown earlier in Section 5.6, the quantified clause Φ that encodes φ's orbit is algorithmically constructed by a syntactic analysis of φ's structure and can involve complex universal and existential quantifier alternations over both state and non-state (auxiliary) variables.

6.2 Range Boosting

Clause boosting is not limited to clauses that are parameterized by the constants of symmetric sorts and can be extended to clauses whose literals depend on the constants of totally-ordered sorts such as ballot, round, epoch, etc., that are used to model the temporal order of events in a distributed protocol. However, the boosting procedure for such clauses differs from symmetric boosting in two ways: a) the ordering relation between totally-ordered constants must be explicitly preserved, and b) adherence of a boosted clause to the unreachability constraint is not guaranteed and must be explicitly checked with a 1-step backward reachability query.

We extended IC3PO with a *range boosting* procedure that complements its symmetry boosting mechanism, allowing it to transparently handle protocols with both symmetric and totally-ordered sorts.

Let φ be a clause that is parameterized by totally-ordered constants and let $\varphi^{Ordered}$ denote those variants of φ that are obtained by ordering-compliant permutations of its constants. Clause φ is boosted by making 1-step backward reachability queries on $\varphi^{Ordered}$ to identify its *safe* subset φ^{Safe}, i.e., those variants that satisfy the unreachability constraint.

For example, consider the following clause φ_1 defined on the finite instance $\mathcal{P}(3, 4)$ from (6.1):

$$\varphi_1 = p(\mathtt{a}_1, \mathtt{b}_1) \lor q(\mathtt{b}_2) \tag{6.2}$$

Since φ_1 contains two ordered constants $(\mathtt{b}_1, \mathtt{b}_2)$, it has six ordering-compliant variants $(\mathtt{b}_{\mathtt{min}}, \mathtt{b}_1)$, $(\mathtt{b}_{\mathtt{min}}, \mathtt{b}_2)$, $(\mathtt{b}_{\mathtt{min}}, \mathtt{b}_{\mathtt{max}})$, $(\mathtt{b}_1, \mathtt{b}_2)$, $(\mathtt{b}_1, \mathtt{b}_{\mathtt{max}})$, and $(\mathtt{b}_2, \mathtt{b}_{\mathtt{max}})$. However only three of these variants end up satisfying the unreachability constraint yielding the following safe subset of $\varphi_1^{Ordered}$:

$$\begin{aligned}\varphi_1^{Safe} = {} & [\; p(\mathtt{a_1}, \mathtt{b_1}) \vee q(\mathtt{b_2}) \quad] \wedge \\ & [\; p(\mathtt{a_1}, \mathtt{b_1}) \vee q(\mathtt{b_{max}}) \;] \wedge \\ & [\; p(\mathtt{a_1}, \mathtt{b_2}) \vee q(\mathtt{b_{max}}) \;]\end{aligned} \tag{6.3}$$

The inferred quantified clause that encodes these three clauses is now constructed using two universally-quantified variables $X_1, X_2 \in \mathtt{bSort_4}$ that replace $\mathtt{b_1}$ and $\mathtt{b_2}$ in φ_1 and expressed as an implication whose antecedent specifies a constraint over the ordered "range" $\mathtt{b_{min}} < X_1 < X_2$ that must be satisfied by the quantified variables:

$$\begin{aligned}\Phi_1 = {} & \forall X_1, X_2 \in \mathtt{bSort_4} : \\ & (\mathtt{b_{min}} < X_1) \wedge (X_1 < X_2) \rightarrow [\; p(\mathtt{a_1}, X_1) \vee q(X_2) \;]\end{aligned} \tag{6.4}$$

In general, a clause that is parameterized by k constants from a totally-ordered domain whose size is greater than k can be range-boosted and encoded by a universally-quantified predicate with k variables which is expressed as an implication whose antecedent is a range constraint that evaluates to true for just those combinations of the k variables that correspond to safe variants of φ.

This procedure extends easily to the case of multiple totally-ordered domains as well, allowing range boosting to be performed independently for each such domain in *any* order since constants from different domains do not interfere with each other.

6.3 Hierarchical Strengthening

As advocated in [163], hierarchical structuring is an effective way to manage complexity during manual proof development. It can also be easily incorporated in the IC3PO style of invariant generation based on symmetry and range boosting.

Given a low-level specification L that implements a high-level specification H, i.e., $L \prec H$, hierarchical strengthening starts by automatically deriving strengthening assertions $H!A^H$ that, together with the safety property $H!P$, proves the safety of H. It then maps and propagates $H!A^H$ to L, denoted as $L!A^H$, and proceeds to prove the strengthened property $L!P \wedge L!A^H$ in L by deriving any additional assertions $L!A^L$ needed to establish the safety of L. The underlying assumption in this procedure is that proving H is much easier than proving L directly, and that any assertions derived to prove H are also applicable, with suitable mapping, to L. The final inductive invariant that proves L will, thus, have the form $L!inv = (L!P \wedge L!A^H) \wedge L!A^L$ which can be interpreted as reducing the complexity of L's proof by strengthening its safety property with assertions derived

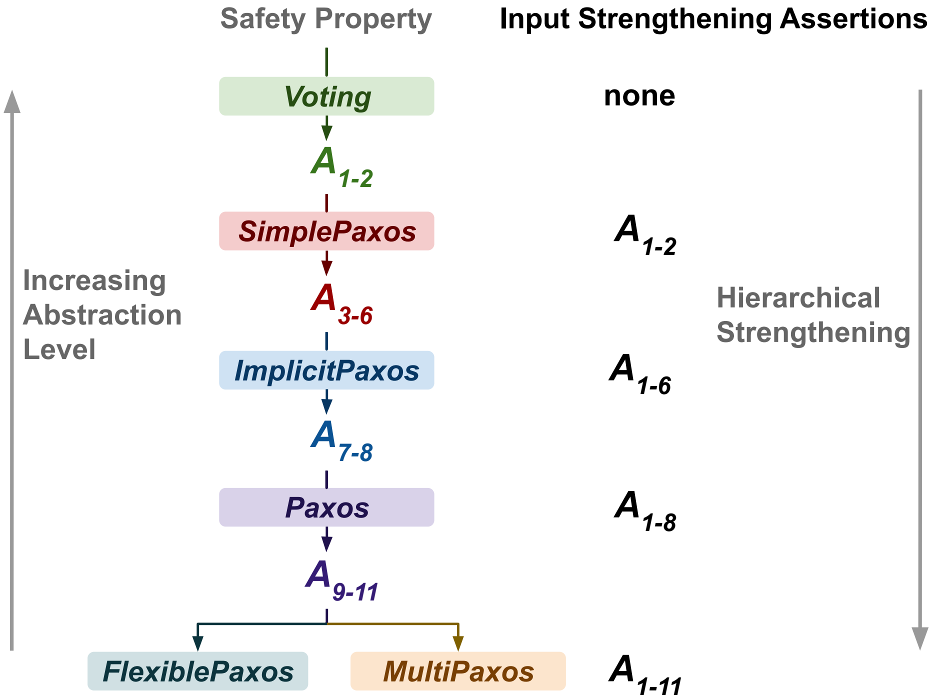


Figure 6.1: Hierarchical strengthening of Paxos and its variants.
Each level uses all strengthening assertions above that level as input, and outputs the required remaining assertions, altogether inferring the inductive invariant at each level.

for H.

Such strengthening can be extended to a k-level hierarchy $H \prec M_1 \prec \cdots \prec M_{k-2} \prec L$, where M_1 to M_{k-2} are suitably defined intermediate levels between H and L. This, in turn, allows single-level automatic verification techniques based on incremental induction, like IC3PO, to scale to complex protocols like *Paxos*, by stepwise verifying higher-level abstractions first and using their auto-generated proofs to incrementally build the proof for the lower-level protocol.

6.4 Hierarchical Specification of Paxos

This section describes in detail a four-level hierarchical structure of the Paxos protocol, as shown in Figure 6.1.

6.4.1 Lamport's *Voting* Protocol

Algorithm 4 presents the TLA+ [168] description[1] of the *Voting* protocol [173], which is a very high-level abstraction of *Paxos* that formalizes the way Lamport first thought about the Paxos

[1] Lamport's TLA+ encoding uses sets to denote variables. For example, in [173], $votes[a]$ represents the set of votes cast by acceptor a. Throughout this chapter, we use an equivalent representation based on relations/functions to enable encoding for SMT solving. $\langle b, v\rangle \in votes[a]$ is equivalently encoded in relational form as $votes(a, b, v) = \top$.

Algorithm 4 Lamport's *Voting* protocol in pretty-printed TLA+

```
CONSTANTS $\mathtt{value}, \mathtt{acceptor}, \mathtt{quorum}$
$\mathtt{ballot} \triangleq Nat \cup \{-1\}$
VARIABLES $votes, maxBal$
$votes \in (\mathtt{acceptor} \times \mathtt{ballot} \times \mathtt{value}) \rightarrow \mathrm{BOOLEAN}$
$maxBal \in \mathtt{acceptor} \rightarrow \mathtt{ballot}$
ASSUME $\land\ \forall\, Q \in \mathtt{quorum} : Q \subseteq \mathtt{acceptor}$
       $\land\ \forall\, Q_1, Q_2 \in \mathtt{quorum} : Q_1 \cap Q_2 \neq \{\}$
$chosenAt(b, v) \triangleq \exists\, Q \in \mathtt{quorum} : \forall\, A \in Q : votes(A, b, v)$
$chosen(v) \triangleq \exists\, B \in \mathtt{ballot} : chosenAt(B, v)$
$showsSafeAt(q, b, v) \triangleq$
  $\land\ \forall\, A \in q : maxBal(A) \geq b$
  $\land\ \exists\, C \in \mathtt{ballot} :$
      $\land\ (C < b)$
      $\land\ (C \neq -1) \rightarrow \exists\, A \in q : votes(A, C, v)$
      $\land\ \forall\, D \in \mathtt{ballot} : (C < D < b) \rightarrow (\forall\, A \in Q : \forall\, V \in \mathtt{value} : \neg votes(A, D, V))$
$isSafeAt(b, v) \triangleq \exists\, Q \in \mathtt{quorum} : showsSafeAt(Q, b, v)$
$IncreaseMaxBal(a, b) \triangleq$
  $\land\ b \neq -1\ \land\ b > maxBal(a)$
  $\land\ maxBal' = [maxBal$ EXCEPT $![a] = b]$
  $\land$ UNCHANGED $votes$
$VoteFor(a, b, v) \triangleq$
  $\land\ b \neq -1\ \land\ maxBal(a) \leq b$
  $\land\ \forall\, V \in \mathtt{value} : \neg votes(a, b, V)$
  $\land\ \forall\, C \in \mathtt{acceptor} : (C \neq a) \rightarrow (\forall\, V \in \mathtt{value} : votes(C, b, V) \rightarrow (V = v))$
  $\land\ isSafeAt(b, v)$
  $\land\ votes' = [votes$ EXCEPT $![a, b, v] = \top]$
  $\land\ maxBal' = [maxBal$ EXCEPT $![a] = b]$
$Init \triangleq$ $\land\ \forall\, A \in \mathtt{acceptor} : B \in \mathtt{ballot} : V \in \mathtt{value} : \neg votes(A, B, V)$
         $\land\ \forall\, A \in \mathtt{acceptor} : maxBal(A) = -1$
$T \triangleq \exists\, A \in \mathtt{acceptor}, B \in \mathtt{ballot}, V \in \mathtt{value} :$
                    $IncreaseMaxBal(A, B) \lor VoteFor(A, B, V)$
$P \triangleq \forall\, V_1, V_2 \in \mathtt{value} : chosen(V_1) \land chosen(V_2) \rightarrow V_1 = V_2$
```

consensus algorithm without getting distracted by details introduced by having the processes communicate by messages. *Voting* has three unordered sorts named `value`, `acceptor` and `quorum`, and a totally-ordered sort named `ballot`. The protocol has two state symbols, $votes$ and $maxBal$ defined on these sorts that serve as the protocol's state variables. $votes(a, b, v)$ is true iff an acceptor a has voted for value v in ballot number b. $maxBal(a)$ returns a ballot number such that acceptor a will never cast any further vote in a ballot numbered less than $maxBal(a)$. The global axiom (line 5) defines the elements of the `quorum` sort to be subsets of the `acceptor` sort and restricts them further by requiring them to be pair-wise non-disjoint. Lines 6-9 specify definitions $chosenAt$, $chosen$, $showsSafeAt$, and $isSafeAt$, which serve as auxiliary non-state variables. Protocol transitions are specified by the actions $IncreaseMaxBal$ and $VoteFor$ (lines 10-11), and lines 12-14 specify the protocol's initial states, transition relation, and safety property.

Viewed as a parameterized system, the template of the *Voting* protocol is *Voting*(`value`, `acceptor`, `quorum`, `ballot`). Its finite instance:

$$
\begin{aligned}
&\textit{Voting}(2,3,3,4):\\
&\quad \mathtt{value}_2 \triangleq \{\mathtt{v}_1, \mathtt{v}_2\}\\
&\quad \mathtt{acceptor}_3 \triangleq \{\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3\}\\
&\quad \mathtt{quorum}_3 \triangleq \{\mathtt{q}_{12}\!:\!\{\mathtt{a}_1, \mathtt{a}_2\},\ \mathtt{q}_{13}\!:\!\{\mathtt{a}_1, \mathtt{a}_3\},\ \mathtt{q}_{23}\!:\!\{\mathtt{a}_2, \mathtt{a}_3\}\}\\
&\quad \mathtt{ballot}_4 \triangleq [\mathtt{b}_{\mathtt{min}}, \mathtt{b}_1, \mathtt{b}_2, \mathtt{b}_{\mathtt{max}}]
\end{aligned}
$$

has three finite symmetric sorts named $\mathtt{value}_2$, $\mathtt{acceptor}_3$ and $\mathtt{quorum}_3$, defined as sets of arbitrarily-named distinct constants, while the finite totally-ordered sort $\mathtt{ballot}_4$ is composed of a list of ordered constants, i.e., $\mathtt{b}_{\mathtt{min}} < \mathtt{b}_1 < \mathtt{b}_2 < \mathtt{b}_{\mathtt{max}}$, where $\mathtt{b}_{\mathtt{min}} = -1$ since -1 is the "minimum" ballot number. The constants of the $\mathtt{quorum}_3$ sort are subsets of the $\mathtt{acceptor}_3$ sort and are named to reflect their symmetric dependence on the $\mathtt{acceptor}_3$ sort. This instance has 24 $votes$ state variables that return a BOOLEAN and 3 $maxBal$ state variables that return a ballot number in $\mathtt{ballot}_4$. A *state* of this instance corresponds to a complete assignment to these 27 state variables.

6.4.2 Lamport's *Paxos* Protocol

Algorithm 5 presents the TLA+ description of Lamport's *Paxos* protocol [171], which is a specification of the Paxos consensus algorithm [165, 167]. *Paxos* implements *Voting* through the refinement mapping $[votes \leftarrow msg2b,\ maxBal \leftarrow maxBal]$, where acceptors now communicate with each other through distributed message passing. State variables $msg1a$, $msg1b$, $msg2a$, and $msg2b$ are used to model the set of different messages that can be sent in the protocol, corresponding to actions $Phase1a$, $Phase1b$, $Phase2a$, and $Phase2b$ respectively. The pair $\langle maxVBal(a), maxVal(a)\rangle$ is the vote with the largest ballot number cast by acceptor a. The ballot b leader can send a $msg1a(b)$ by performing the action $Phase1a(b)$. $Phase1b(a, b)$ implements the $IncreaseMaxBal(a, b)$ action from *Voting*, where after receiving $msg1a(b)$, acceptor a

Algorithm 5 Lamport's *Paxos* protocol in pretty-printed TLA+

```
CONSTANTS value, acceptor, quorum
ballot $\triangleq$ $Nat \cup \{-1\}$
VARIABLES $msg1a, msg1b, msg2a, msg2b, maxBal, maxVBal, maxVal$
$msg1a$    $\in$ ballot $\rightarrow$ BOOLEAN
$msg1b$    $\in$ (acceptor $\times$ ballot $\times$ ballot $\times$ value) $\rightarrow$ BOOLEAN
$msg2a$    $\in$ (ballot $\times$ value) $\rightarrow$ BOOLEAN
$msg2b$    $\in$ (acceptor $\times$ ballot $\times$ value) $\rightarrow$ BOOLEAN
$maxBal$   $\in$ acceptor $\rightarrow$ ballot
$maxVBal$  $\in$ acceptor $\rightarrow$ ballot
$maxVal$   $\in$ acceptor $\rightarrow$ value
$none$     $\in$ value
ASSUME $\wedge\ \forall\, Q \in$ quorum : $Q \subseteq$ acceptor
       $\wedge\ \forall\, Q_1, Q_2 \in$ quorum : $Q_1 \cap Q_2 \neq \{\}$
$chosenAt(b, v)$ $\triangleq$ $\exists\, Q \in$ quorum : $\forall\, A \in Q : msg2b(A, b, v)$
$chosen(v)$ $\triangleq$ $\exists\, B \in$ ballot : $chosenAt(B, v)$
$showsSafeAtPaxos(q, b, v)$ $\triangleq$
  $\wedge\ \forall\, A \in q : \exists\, M_b \in$ ballot : $\exists\, M_v \in$ value : $msg1b(A, b, M_b, M_v)$
  $\wedge\ \vee\ \forall\, A \in$ acceptor : $\forall\, M_b \in$ ballot : $\forall\, M_v \in$ value :
              $\neg(\, A \in q\ \wedge\ msg1b(A, b, M_b, M_v)\ \wedge\ (M_b \neq\ -1)\,)$
     $\vee\ \exists\, M_b \in$ ballot :
        $\wedge\ \exists\, A \in q : msg1b(A, b, M_b, v)\ \wedge\ (M_b \neq\ -1)$
        $\wedge\ \forall\, A \in q : \forall\, M_{b2} \in$ ballot : $\forall\, M_{v2} \in$ value :
              $msg1b(A, b, M_{b2}, M_{v2})\ \wedge\ (M_{b2} \neq\ -1) \rightarrow M_{b2} \leq M_b$
$isSafeAtPaxos(b, v)$ $\triangleq$ $\exists\, Q \in$ quorum : $showsSafeAtPaxos(Q, b, v)$
$Phase1a(b)$ $\triangleq$
  $\wedge\ b \neq\ -1$
  $\wedge\ msg1a' = [msg1a$ EXCEPT $![b] = \top]$
  $\wedge$ UNCHANGED $msg1b, msg2a, msg2b, maxBal, maxVBal, maxVal$
$Phase1b(a, b)$ $\triangleq$
  $\wedge\ b \neq\ -1\ \wedge\ msg1a(b)\ \wedge\ b > maxBal(a)$
  $\wedge\ maxBal' = [maxBal$ EXCEPT $![a] = b]$
  $\wedge\ msg1b' = [msg1b$ EXCEPT $![a, b, maxVBal(a), maxVal(a)] = \top]$
  $\wedge$ UNCHANGED $msg1a, msg2a, msg2b, maxVBal, maxVal$
```

(continued)

12 $Phase2a(b, v) \triangleq$
$\quad \land\ b \neq -1 \ \land\ v \neq none \ \land\ \neg(\exists\, V \in \mathtt{value} : msg2a(b, V))$
$\quad \land\ isSafeAtPaxos(b, v)$
$\quad \land\ msg2a' = [msg2a \text{ EXCEPT } ![b, v] = \top]$
$\quad \land\ \text{UNCHANGED } msg1a, msg1b, msg2b, maxBal, maxVBal, maxVal$

13 $Phase2b(a, b, v) \triangleq$
$\quad \land\ b \neq -1 \ \land\ v \neq none \ \land\ msg2a(b, v) \ \land\ b \geq maxBal(a)$
$\quad \land\ maxBal' \ = [maxBal \text{ EXCEPT } ![a] = b]$
$\quad \land\ maxVBal' = [maxVBal \text{ EXCEPT } ![a] = b]$
$\quad \land\ maxVal' \ = [maxVal \text{ EXCEPT } ![a] = v]$
$\quad \land\ msg2b' \ = [msg2b \text{ EXCEPT } ![a, b, v] = \top]$
$\quad \land\ \text{UNCHANGED } msg1a, msg1b, msg2a$

14 $Init \triangleq \forall\, A \in \mathtt{acceptor} : B \in \mathtt{ballot} :$
$\qquad \land\ \neg msg1a(B)$
$\qquad \land\ \forall\, M_b \in \mathtt{ballot} : M_v \in \mathtt{value} : \neg msg1b(A, B, M_b, M_v)$
$\qquad \land\ \forall\, V \in \mathtt{value} : \neg msg2a(B, V) \ \land\ \neg msg2b(A, B, V)$
$\qquad \land\ maxBal(A) = -1$
$\qquad \land\ maxVBal(A) = -1 \ \land\ maxVal(A) = none$

15 $T \triangleq \exists\, A \in \mathtt{acceptor} : B \in \mathtt{ballot} : V \in \mathtt{value} :$
$\qquad \lor\ Phase1a(B) \qquad \lor\ Phase1b(A, B)$
$\qquad \lor\ Phase2a(B, V) \ \lor\ Phase2b(A, B, V)$

16 $P \triangleq \forall\, V_1, V_2 \in \mathtt{value} : chosen(V_1) \land chosen(V_2) \rightarrow V_1 = V_2$

sends $msg1b$ to the ballot b leader containing the values of $maxVBal(a)$ and $maxVal(a)$. In the $Phase2a(b, v)$ action, the ballot b leader sends $msg2a$ asking the acceptors to vote for a value v that is safe at ballot number b. Its enabling condition $isSafeAtPaxos(b, v)$ checks the enabling condition $isSafeAt(b, v)$ from *Voting*. $Phase2b$ implements the $VoteFor$ action in *Voting*, and enables acceptor a to vote for value v in ballot number b. We refer the reader to [172] for a detailed explanation to understand the internals of *Paxos*.

Represented as a parameterized system *Paxos*(`value`, `acceptor`, `quorum`, `ballot`), its finite instance *Paxos*$(2, 3, 3, 4)$ has 132 BOOLEAN state variables, 6 state variables that return a ballot number in $\mathtt{ballot}_4$, and 3 state variables that return a value in $\mathtt{value}_2$.

6.4.3 Intermediate Levels between *Voting* and *Paxos*

We introduced two intermediate levels, *SimplePaxos* and *ImplicitPaxos*, between *Voting* and *Paxos* (Appendix B.2). These intermediate levels are abstractions of *Paxos*, inspired from the already-existing literature [169, 213, 211, 15, 183]. *ImplicitPaxos* is inspired from the specification of Generalized Paxos by Lamport [169] and uses a commonly used encoding transformation, as utilized

Algorithm 5 Modifications in *ImplicitPaxos* compared to *Paxos*

```
11 $Phase1b(a, b) \triangleq$
     $\wedge\ b \neq -1 \ \wedge\ msg1a(b) \ \wedge\ b > maxBal(a)$
     $\wedge\ maxBal' = [maxBal \text{ EXCEPT } ![a] = b]$
     $\wedge\ \exists\, M_b \in \texttt{ballot} : \exists\, M_v \in \texttt{value} :$
         $\wedge\ \ \vee\ \ \wedge\ (M_b = -1)$
                      $\wedge\ \forall\, B \in \texttt{ballot} : \forall\, V \in \texttt{value} : \neg msg2b(a, B, V)$
              $\vee\ \ \wedge\ (M_b \neq -1) \wedge msg2b(a, M_b, M_v)$
                      $\wedge\ \forall\, B \in \texttt{ballot} : \forall\, V \in \texttt{value} : msb2b(a, B, V) \rightarrow B \leq M_b$
         $\wedge\ msg1b' = [msg1b \text{ EXCEPT } ![a, b, M_b, M_v] = \top]$
     $\wedge\ \text{UNCHANGED } msg1a, msg2a, msg2b$
```

Algorithm 6 Modifications in *SimplePaxos* compared to *ImplicitPaxos*

```
8 $showsSafeAtSimplePaxos(q, b, v) \triangleq$
    $\wedge\ \forall\, A \in q : msg1b(A, b)$
    $\wedge\ \ \vee\ \forall\, A \in \texttt{acceptor} : \forall\, M_b \in \texttt{ballot} : \forall\, M_v \in \texttt{value} :$
                    $\neg(\, A \in q \ \wedge\ msg1b(A, b) \ \wedge\ msg2b(A, M_b, M_v)\,)$
       $\vee\ \exists\, M_b \in \texttt{ballot} :$
         $\wedge\ \exists\, A \in q : msg1b(A, b) \ \wedge\ msg2b(A, M_b, v)$
         $\wedge\ \forall\, A \in q : \forall\, M_{b2} \in \texttt{ballot} : \forall\, M_{v2} \in \texttt{value} :$
                    $msg1b(A, b) \ \wedge\ msg2b(A, M_{b2}, M_{v2}) \rightarrow M_{b2} \leq M_b$
```

in [15, 211, 183]. Instead of explicitly keeping a track of $maxVBal(a)$ and $maxVal(a)$, *ImplicitPaxos* abstracts them away and implicitly computes their respective values using the history of all votes cast by the acceptor a, i.e., using the history of $msg2b$ from acceptor a, by modifying the $Phase1b(a, b)$ action (line 11 in Algorithm 5) to as shown in Algorithm 5.

SimplePaxos further simplifies *ImplicitPaxos* and eliminates tracking of the maximum ballot (and the corresponding value) in which an acceptor voted from $msg1b$ completely, i.e., the last two arguments of $msg1b$ are abstracted away. Instead, the history of all votes cast is used to describe how new votes are cast. This is done by replacing the definition $showsSafeAtPaxos$ (line 8 in Algorithm 5) with its simplified form, expressed using $msg2b$ as shown in Algorithm 6.

6.5 Hierarchical Verification of Paxos

Using the 4-level hierarchy *Paxos* $\prec$ *ImplicitPaxos* $\prec$ *SimplePaxos* $\prec$ *Voting*, this section is a "log" of how IC3PO automatically derived the required strengthening assertions that established the safety of *Paxos*.

6.5.1 Proving *Voting*

Using instance *Voting*$(2, 3, 3, 4)$, IC3PO proved the safety of *Voting* by automatically deriving the inductive invariant $V!inv \triangleq V!P \wedge V!A_1 \wedge V!A_2$ where

$$V!A_1 = \forall A \in \texttt{acceptor}, B \in \texttt{ballot}, V \in \texttt{value} : \; votes(A, B, V) \rightarrow isSafeAt(B, V)$$

$$V!A_2 = \forall A \in \texttt{acceptor}, B \in \texttt{ballot}, V_1, V_2 \in \texttt{value} : \quad chosenAt(B, V_1) \wedge votes(A, B, V_2) \rightarrow (V_1 = V_2)$$

In words, these two strengthening assertions mean:

A_1: If an acceptor voted for value V in ballot number B, then V is safe at B.

A_2: If value V_1 is chosen at ballot B, then no acceptor can vote for a value different than V_1 in B.

6.5.2 Proving *SimplePaxos*

Using the refinement mapping $[votes \leftarrow msg2b, \; maxBal \leftarrow maxBal]$, IC3PO transformed $V!A_1$ and $V!A_2$ to the following corresponding versions for *SimplePaxos*:

$$S!A_1 = \forall A \in \texttt{acceptor}, B \in \texttt{ballot}, V \in \texttt{value} : \; msg2b(A, B, V) \rightarrow isSafeAt(B, V)$$

$$S!A_2 = \forall A \in \texttt{acceptor}, B \in \texttt{ballot}, V_1, V_2 \in \texttt{value} : \quad chosenAt(B, V_1) \wedge msg2b(A, B, V_2) \rightarrow (V_1 = V_2)$$

These two assertions, passed down from the proof of *Voting*, represented a strengthening of the safety property of *SimplePaxos* that allowed IC3PO to prove it with the inductive invariant $S!inv \triangleq S!P \wedge \bigwedge_{1 \leq i \leq 6} S!A_i$ where

$$S!A_3 = \forall B \in \texttt{ballot}, V \in \texttt{value} : \; msg2a(B, V) \rightarrow isSafeAt(B, V)$$

$$S!A_4 = \forall B \in \texttt{ballot}, V_1, V_2 \in \texttt{value} : \; msg2a(B, V_1) \wedge msg2a(B, V_2) \rightarrow (V_1 = V_2)$$

$$S!A_5 = \forall A \in \texttt{acceptor}, B \in \texttt{ballot}, V \in \texttt{value} : \; msg2b(A, B, V) \rightarrow msg2a(B, V)$$

$$S!A_6 = \forall A \in \texttt{acceptor}, B \in \texttt{ballot} : \; msg1b(A, B) \rightarrow maxBal(A) \geq B$$

are four additional automatically-generated strengthening assertions that express the following facts about *SimplePaxos*:

A_3: If ballot B leader sends a $2a$ message for value V, then V is safe at B.

A_4: A ballot leader can send $2a$ messages only for a unique value.

A_5: If an acceptor voted for a value in ballot number B, then there is a $2a$ message for that value at B.

A_6: If an acceptor has sent a $1b$ message at a ballot number B, then its $maxBal$ is at least as high as B.

6.5.3 Proving *ImplicitPaxos*

All variables from *SimplePaxos* refine to *ImplicitPaxos* as is, except for $msg1b$ that adds explicit tracking of the maximum vote voted by an acceptor in *ImplicitPaxos*. Assertions $S!A_1$ to $S!A_5$ map to $I!A_1$ to $I!A_5$ in *ImplicitPaxos* as is, while $S!A_6$ maps as:

$$I!A_6 = \forall A \in \texttt{acceptor}, B, B_{max} \in \texttt{ballot}, V_{max} \in \texttt{value} : \\ msg1b(A, B, B_{max}, V_{max}) \rightarrow maxBal(A) \geq B$$

These six assertions, passed down from the proof of *SimplePaxos*, represented a strengthening of the safety property of *ImplicitPaxos* that allowed IC3PO to prove it with the inductive invariant $I!inv \triangleq I!P \wedge \bigwedge_{1 \leq i \leq 8} I!A_i$ where

$$I!A_7 = \forall A \in \texttt{acceptor}, B, B_{max} \in \texttt{ballot}, V_{max} \in \texttt{value} : \\ [(B > -1) \wedge (B_{max} > -1) \wedge msg1b(A, B, B_{max}, V_{max})] \rightarrow msg2b(A, B_{max}, V_{max})$$

$$I!A_8 = \forall A \in \texttt{acceptor}, B, B_{mid}, B_{max} \in \texttt{ballot}, V, V_{max} \in \texttt{value} : \\ [(B > B_{mid}) \wedge (B_{mid} > B_{max}) \wedge msg1b(A, B, B_{max}, V_{max})] \rightarrow \neg msg2b(A, B_{mid}, V)$$

are two additional automatically-generated strengthening assertions that express the following facts about *ImplicitPaxos*:

A_7: If an acceptor issued a $1b$ message at ballot number B with the maximum vote $\langle B_{max}, V_{max} \rangle$, and both B and B_{max} are higher than -1, then the acceptor has voted for value V_{max} in ballot B_{max}.

A_8: If an acceptor issued a $1b$ message at ballot number B with the maximum vote $\langle B_{max}, V_{max} \rangle$, then the acceptor cannot have voted in any ballot number strictly between B_{max} and B.

6.5.4 Proving *Paxos*

All variables from *ImplicitPaxos* refine to *Paxos* trivially, mapping $I!A_1, \ldots, I!A_8$ to $P!A_1, \ldots, P!A_6$ in *Paxos* as is. These eight assertions, passed down from the proof of *ImplicitPaxos*, represented a strengthening of the safety property of *Paxos* that allowed IC3PO to prove it with the inductive invariant $P!inv \triangleq P!P \wedge \bigwedge_{1 \leq i \leq 11} P!A_i$ where

$$P!A_9 = \forall A \in \texttt{acceptor} : \; maxVBal(A) \leq maxBal(A)$$

$$P!A_{10} = \forall A \in \texttt{acceptor}, B \in \texttt{ballot}, V \in \texttt{value} : \; msg2b(A, B, V) \rightarrow maxVBal(A) \geq B$$

$$P!A_{11} = \forall A \in \texttt{acceptor} : \; maxVBal(A) > -1 \rightarrow msg2b(A, maxVBal(A), maxVal(A))$$

are three additional automatically-generated strengthening assertions that express the following facts about *Paxos*:

A_9: $maxVBal$ of an acceptor is less than or equal to its $maxBal$.

A_{10}: If an acceptor voted in a ballot number B, then its $maxVBal$ is at least as high as B.

A_{11}: If acceptor A has its $maxVBal$ higher than -1, then A has already cast a vote $\langle maxVBal(A), maxVal(A) \rangle$.

6.6 Discussion

This section provides a discussion about certain key points and features about the *Paxos* proof from Section 6.5.

6.6.1 Comparison against Human-written Invariants

Optionally, the inductive invariant $P!inv$ can be minimized to derive a subsumption-free and closed set of invariants, which removes A_1 and A_2 that are subsumed by the conjunction $A_3 \wedge A_4 \wedge A_5$. After this minimization, the inductive invariant of *Paxos* matches identically with the manually-written and TLAPS-checked inductive invariant from [103], guaranteeing its correctness. Similarly, the inductive invariant of *Voting*, i.e., $V!inv$, matches directly with the manually-written and TLAPS-checked inductive invariant from [174].

6.6.2 Benefits of Range Boosting

Assertions A_6 to A_{11} express conditions defined over ordered ranges in the *infinite* totally-ordered `ballot` domain. Inferring such invariants automatically through IC3PO becomes possible through range boosting (Section 6.2), that extends incremental induction with the knowledge of *temporal regularity* over totally-ordered domains by learning quantified clauses over ordered ranges.

6.6.3 Protocol's Formula Structure

Note that A_1 to A_3 use definitions $isSafeAt$ and $chosenAt$, which implicitly enables IC3PO to incorporate learning with complex quantifier alternations. Inspired from previous works on the importance of using derived/ghost variables [161, 209, 202], IC3PO utilizes the *formula structure* of the protocol's transition relation in a unique manner, by incorporating *definitions* in the protocol specification as auxiliary non-state variables during reachability analysis. This provides a simple and inexpensive procedure to incorporate clause learning with complex quantifier alternations.

6.6.4 Decidability

Protocol specifications at each of the four levels include quantifier alternation cycles that make unbounded SMT reasoning fall into the undecidable fragment of first-order logic. Unsurprisingly, previous works that rely on unbounded SMT reasoning, like SWISS [136], fol-ic3 [153], DistAI [235], I4 [186], and UPDR [152], struggle with verifying Lamport's Paxos. IC3PO, on the other hand, performs incremental induction and finite convergence over finite protocol instances using finite-domain reasoning that is always decidable.

6.6.5 Why a Four-Level Hierarchy?

The original Paxos specification is composed of a two-level hierarchy *Paxos* $\prec$ *Voting*. Given the two strengthening assertions A_1 and A_2 from *Voting*, inferring the remaining nine assertions for *Paxos* directly in one step of hierarchical strengthening is difficult, since these two specifications are too far apart to be proved directly. IC3PO struggled with the large state space of *Paxos* and learnt too many weak clauses involving $msg1b$, $maxVBal$ and $maxVal$, eventually running out of memory due to invariant inference getting confused with several counterexamples-to-induction. Table 6.1 compares the state-space size of protocol instances at each of the four hierarchical levels. Even though 2^{147} is not huge, especially with respect to hardware verification problems [52, 120, 124], *Paxos* has a dense state-transition graph where state-transitions are tightly coupled with high in- and out- degree, making the problem difficult for automatic invariant inference with incremental induction.

Adding *ImplicitPaxos* reduced the complexity in *Paxos* by abstracting away $maxVBal$ and $maxVal$. Still, scalability remained a challenge due to $msg1b$, that contributed to 96 out of 147 state bits in *Paxos*$(2, 3, 3, 4)$. Adding another level, i.e., *SimplePaxos*, removed 84 out of these 96 state bits by abstracting away explicit tracking of the maximum vote of an acceptor from $msg1b$. When compared against *Paxos*, *SimplePaxos* is significantly simpler, with a total state-space size to be just 2^{54} for its finite instance *SimplePaxos*$(2, 3, 3, 4)$, which led IC3PO to successfully prove *Paxos* automatically using the four-level hierarchy.

6.6.6 Extension to *MultiPaxos* and *FlexiblePaxos*

Till now, by *Paxos* we meant *single-decree* Paxos which is the core consensus algorithm underlying the complete Paxos state-machine replication protocol [165, 167], commonly referred to as *MultiPaxos* [15]. In *MultiPaxos*, a sequence of instances executes single-decree *Paxos* such that the value chosen in the i^{th} instance becomes the i^{th} command executed by the replicated state machine. Additionally, if the leader is relatively stable, $Phase1$ becomes unnecessary and is skipped, reducing the failure-free message delay from 4 delays to 2 delays.

Mapping each of the assertions $A_1, \ldots, A_{11}$ to *MultiPaxos* is trivial, and simply adds the corresponding instance as an additional universally-quantified argument, e.g., A_{11} maps as:

$$M!A_{11} = \forall A \in \texttt{acceptor}, I \in \texttt{instances} : \\ \qquad maxVBal(A, I) > -1 \rightarrow msg2b(A, I, maxVBal(A, I), maxVal(A, I))$$

Unsurprisingly, the 11 strengthening assertions, passed down from the proof of *Paxos*, together with the safety property of *MultiPaxos*, allowed IC3PO to trivially prove it with no additional strengthening assertions needed, meaning $M!P \wedge \bigwedge_{1 \leq i \leq 11} M!A_i$ is already an inductive invariant of *MultiPaxos*. As described in previous works [101, 165, 167, 73], the crux of proving the safety of *MultiPaxos* is based on proving single-decree *Paxos* since each consensus instance participates independently without any interference from other instances. Our experiments validated this further.

Similarly, we also tried another Paxos variant called *FlexiblePaxos* [145], which also verifies trivially with the same inductive invariant, i.e., with no additional strengthening assertions needed.

Finite Instance	State-space Size
Voting$(2, 3, 3, 4)$	2^{30}
SimplePaxos$(2, 3, 3, 4)$	2^{54}
ImplicitPaxos$(2, 3, 3, 4)$	2^{138}
Paxos$(2, 3, 3, 4)$	2^{147}

Table 6.1: State-space size for finite instances with 2 `value`, 3 `acceptor`, 3 `quorum`, and 4 `ballot`

6.7 Experiments

	Protocol	S.A.	IC3PO	SWISS	fol-ic3	DistAI	I4	UPDR
EPR	epr-paxos	∅	**568**	15950*	timeout	unknown	memout	timeout
	epr-flexible_paxos	∅	**561**	18232*	timeout	unknown	memout	unknown
	epr-multi_paxos	∅	timeout	timeout	timeout	unknown	memout	timeout
ORIGINAL	*Voting*	∅	**64**	timeout	timeout	unknown	memout	timeout
	SimplePaxos	A_{1-2}	**51**	timeout	timeout	unknown	unknown	timeout
	ImplicitPaxos	A_{1-6}	**2008**	timeout	timeout	unknown	unknown	timeout
	Paxos	A_{1-8}	**98**	timeout	timeout	unknown	unknown	timeout
	MultiPaxos	A_{1-11}	**340**	timeout	timeout	unknown	timeout	timeout
	FlexiblePaxos	A_{1-11}	**1408**	timeout	timeout	unknown	unknown	timeout

Table 6.2: Comparison of IC3PO against other state-of-the-art verifiers
ORIGINAL problems employ hierarchical strengthening (as detailed in Section 6.5), while EPR problems do not. Column 2 (labeled S.A.) lists strengthening assertions added through hierarchical strengthening to the safety property (∅ means none). Columns 3-8 compare the runtime in seconds. For failed SWISS runs, we include the runtime from [136] (indicated with *).

			Inv		SMT	
	Protocol	S.A.	IC3PO	Human	IC3PO	I4
EPR	epr-paxos	∅	6	11	5680	1701556
	epr-flexible_paxos	∅	6	11	1509	1761504
	epr-multi_paxos	∅	–	12	–	1902621
ORIGINAL	*Voting*	∅	3	3	1057	1714170
	SimplePaxos	A_{1-2}	5	5	618	158470
	ImplicitPaxos	A_{1-6}	7	7	18329	69715
	Paxos	A_{1-8}	10	10	668	76030
	MultiPaxos	A_{1-11}	10	10	161	–
	FlexiblePaxos	A_{1-11}	10	10	161	6983

Table 6.3: Comparison of invariant size and the number of SMT queries
Columns 3-4 (labeled Inv) compare number of assertions in the inductive invariant between IC3PO (with subsumption checking and minimization) and human-written proofs. Columns 5-6 (labeled SMT) compare total number of SMT queries made by IC3PO versus I4 (until failure for unsuccessful runs).

IC3PO [127] currently accepts protocol descriptions in the Ivy language [210] and uses the Ivy compiler to extract a logical formulation of the protocol in a SMT-LIB [46] compatible format. To get an idea on the effectiveness of hierarchical strengthening, we also evaluated automatically deriving inductive proofs for EPR variants of Paxos from [211] without any hierarchical strengthening. These specifications describe Paxos in the EPR fragment [214] of first-order logic and also incorporate simplifications equivalent to the ones described for *SimplePaxos* in Section 6.4.3. We performed a detailed comparison against other state-of-the-art techniques for automatically verifying distributed protocols:

- SWISS [136] uses SMT solving to derive an inductive invariant by performing an enumerative search in an optimized and bounded invariant search space.

- fol-ic3 [153], implemented in *mypyvy* [16], extends IC3 with a separators-based technique that performs enumerative search for a quantified separator in the space of bounded mixed quantifier prefixes.

- DistAI [235] performs data-driven invariant learning by enumerating over possible invariants derived from simulating a protocol at different instance sizes, followed by iteratively refining and checking candidate invariants.

- I4 [186, 187] performs finite-domain IC3 (without accounting for regularity) using the AVR model checker [124, 125], followed by iteratively generalizing and checking the inductive invariant produced by AVR.

- UPDR, from the *mypyvy* [16] framework, implements PDR$^\forall$/UPDR [151] for verifying distributed protocols.

All experiments were performed on an Intel (R) Xeon CPU (X5670). For each run, we used a 5-hour timeout and a 32 GB memory limit. All tools were executed in their respective default configurations. We used Z3 [98] version 4.8.10, Yices 2 [107] version 2.6.2, and CVC4 [45] version 1.8.

6.7.1 Results

Tables 6.2 & 6.3 summarize the experimental results. `EPR` variants were run without any hierarchical strengthening. For `ORIGINAL` problems, we employed hierarchical strengthening using each tool to verify Lamport's original Paxos specification (and its variants) through higher-level strengthening assertions that were automatically generated from IC3PO, as detailed in Section 6.5. Note that `ORIGINAL` problems include quantifier-alternation cycles that make unbounded SMT reasoning fall into the undecidable fragment of first-order logic.

IC3PO emerges as the only successful technique that verifies Lamport's Paxos and its variants, and automatically infers the required inductive invariants efficiently. Unsurprisingly, none of the other tools (i.e., SWISS, fol-ic3, DistAI, I4 and UPDR) were able to solve `ORIGINAL` problems since each of these tools rely on unbounded SMT reasoning and struggle on problems that fall outside the decidable EPR fragment of first-order logic.

6.7.2 Discussion

Effect of hierarchical strengthening: Comparing EPR versus ORIGINAL shows the advantages offered by hierarchical strengthening. Even though IC3PO was able to automatically verify EPR versions of single-decree Paxos and flexible Paxos from [211], none of the tools were able to automatically verify the EPR version of multi-decree Paxos. ORIGINAL variants, on the other hand, employed hierarchical strengthening which allowed IC3PO to verify Lamport's Paxos automatically and efficiently by using the protocol's hierarchical structure.

Comparison against other verifiers: DistAI, I4, and UPDR are limited to generating only universally-quantified invariants over state variables, and hence, were unable to solve any problem. While both IC3PO and I4 use incremental induction over a finite protocol instance, the number of SMT queries made by I4 grows drastically, indicating the benefits offered by symmetry and range boosting employed in IC3PO. fol-ic3 also fails on all problems, showing limited scalability of its enumeration-based separators technique operating directly in the unbounded domain. For SWISS, we couldn't replicate results for EPR problems as reported in [136] using our experimental setup. Nevertheless, SWISS showed limited capabilities for solving ORIGINAL problems.

Comparison against human-written invariants: As evident from A_1 to A_{11} in Section 6.5, IC3PO generated concise, human-readable inductive invariants. In fact, every invariant of *Paxos* written manually by Lamport et al. (as detailed in [171, 103]) had a corresponding equivalent invariant in the inductive proof automatically generated with IC3PO. In contrast, deriving such invariants manually, even in the presence of a hierarchical structure, is a tedious and error-prone process that demands deep domain expertise [103, 138, 211, 226].

Overall, the evaluation confirms our main hypothesis, that it is possible to utilize the regularity and hierarchical structure in complex distributed protocols, like in Paxos, to scale automatic verification beyond the current state-of-the-art.

6.8 Summary

We proposed *range boosting*, a novel technique that extends the incremental induction algorithm to utilize the temporal regularity in distributed protocols through quantified reasoning over ordered ranges. We also presented *hierarchical strengthening*, a simple technique that utilizes the hierarchical structure of protocol specifications to enable automatic verification of complex distributed protocols with high scalability. Given the four-level hierarchy of the Paxos specification, we showed that these techniques, coupled with our recent work on symmetry boosting and finite convergence, provide, to our knowledge, the first demonstration of an automatically-inferred inductive invariant for the original Lamport's Paxos algorithm.

CHAPTER 7

Conclusions and Future Work

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

7.1 Conclusions

Our work extends automated reasoning and abstraction methods enabling the scalable verification of complex hardware designs and distributed protocols. We explored techniques that automate some of the critical steps in design verification that are beyond the scope of existing state-of-the-art techniques, and thus, require profound human efforts and manual intuition from a domain expert in practice. Our techniques build on decades of progress in verification methods to improve scalability and automation in this field. These techniques also guarantee provable correctness and assurance by automatically producing independently-checkable explanations, in the form of proof certificates and counterexample traces, to mathematically justify the verification outcome. Our in-depth evaluation has demonstrated that the developed techniques can automatically verify the correctness of several complex systems, including several interesting real-world designs whose internal workings are too hard to understand for a non-expert.

7.1.1 Hardware Verification

For hardware verification, the key novelties in this dissertation research can be summarized as follows:

- The idea of recognizing relevant information from the structure of the system isn't new, though unlike previous methods, our novel abstraction technique based on equality relations, called *equality abstraction* (EA), allows for a comprehensive representation and an implicit generation of the abstract state space without the need for any manual efforts or expensive computation to identify the most crucial relations among design objects. The abstract state-space size is completely independent of the bit width of variables.

- We developed an effective incremental induction algorithm that operates directly at the design level by integrating EA with a novel refinement procedure and efficient structure-guided cube generalization. Refining EA to eliminate any spurious behavior in the abstract domain is performed without any path explosion and is fully incremental, that allows reusing all clause learning from previous iterations. Our novel cone-of-influence and justification-based procedure performs cube generalization at the word level without requiring any solver calls and without any implicit or explicit unrolling of the transition relation. The resulting WIC3+EA algorithm supports EA with both interpreted and uninterpreted functions and offers scalable model checking of control-centric problems irrespective of the state-space size or complexity of operations.

- We developed a collection of extensions, such as hybrid data abstraction, configuring EA granularity, proof race, etc. to complement WIC3+EA and increase verification scalability for different kinds of problems.

These techniques led to the development of a fully-automatic verifier called AVR [124, 125], and led to successfully solving challenging verification problems from a diverse set of open-source and industrial hardware designs [120, 124, 52], including problems defined on different RISC-V CPU implementations, flash and video display controllers, and industrial multiplier designs. AVR won the Hardware Model Checking Competition in 2020, outperforming existing state-of-the-art model checkers with a wide margin. AVR is available open source on GitHub [121].

7.1.2 Distributed Protocol Verification

For verifying distributed protocols, this dissertation work makes the following contributions:

- We proposed a novel, constructive approach to automatically infer the required quantified inductive invariants for distributed protocols. The technique does not prescribe, a priori, a specific quantifier prefix. Instead, the required prefix is automatically inferred without any enumerative search by carefully analyzing the *spatial* and *temporal* regularity of the protocol. The key insight underlying this approach is that these structural regularities and quantification are closely related concepts that express protocol invariance under different rearrangements of its components or its evolution over time. We extended finite incremental induction to use these regularities to *boost* clause learning by "growing" a single clause φ to a set of clauses that can be safely learned during reachability analysis. Quantifier inference compactly encodes this clause set into a quantified predicate Φ through a simple analysis of φ's *syntactic structure* and yields a quantified form with both universal and existential quantifiers.

- We developed a systematic *finite convergence* procedure for determining a minimal instance size, the *cutoff*, that represents all unique protocol behaviors and whose analysis is sufficient to prove safety in the unbounded case, yielding a quantified proof that proves safety for any size. The intuition behind this is that the set of protocol behaviors increases monotonically with increased instance sizes but eventually *saturates*.
- For hierarchically-specified distributed protocols, we developed *hierarchical strengthening*, a technique that derives the required inductive invariant in a top-down stepwise procedure, by automatically verifying high-level abstractions first and using invariants of these higher-level abstractions as *strengthening assertions* to derive the inductive invariant for the detailed lower-level protocol.

Recognizing these different structural features in distributed protocols, we developed IC3PO [126, 128], a new verifier for distributed protocols that significantly outperforms the state-of-the-art, scales orders of magnitude faster, and derives compact inductive invariants fully automatically. As a significant milestone, we demonstrated how IC3PO was used to perform safety verification of *Lamport's Paxos algorithm*, both single- and multi-decree Paxos, through the derivation of a compact, human-readable inductive proof that is automatically inferred with IC3PO [129, 130], resulting in a drastic reduction in verification effort compared to previous approaches. IC3PO is available open source on GitHub [127].

7.2 Future Work

There are many interesting avenues of future work that follow from the research presented in this dissertation. We discuss some of these ideas in this section.

Fine-tuned Data Abstraction: Data dependency can make word-level techniques in AVR ineffective for certain kinds of problems, especially when a significant number of data operations occurring at different bit widths play an important role in determining the verification outcome. For example, a data-dependent problems defined on a multi-stage pipelined CPU may require complete interpretation of all operations involved in the instruction decode phase to determine whether or not the property holds irrespective of their bit widths, while all other data operations in other phases can be safely abstracted away as uninterpreted. While hybrid data abstraction based on an interpretation threshold, detailed in Section 4.2.2, offers a limited solution, this approach can be fine-tuned to identify a much more precise subset of data operations important to establish the property and better capture the involved data dependency. Instead of abstracting away all data operations as uninterpreted beyond an interpretation threshold, one possible solution is to only apply data abstraction to a selective subset of computationally heavy operations that are deemed irrelevant to the property while retaining the interpretation of all other operations. This can be complemented with a CEGAR-style approach to iteratively identify which operations are important to the property, where the refinement step incrementally re-configures the abstraction by converting critical data operations identified as important during refinement back to interpreted. Another idea is to add partial interpretation to light-weight data operations, like bitwise and shift operations, in the EUF domain, in a similar manner as partial interpretation of bit-field extraction and concatenation explored in Section 4.2.1.

Combined Forward and Backward Reachability: While most techniques presented in this dissertation relate to the state-of-the-art backward reachability analysis using the IC3/PDR algorithm, previous works [137, 219, 220] have shown complementary strengths of its forward counterpart called "Reverse PDR", which starts its analysis with the initial states instead of the unsafe states as in the original PDR. A combined forward-backward version that inherits the advantages of both original and reverse incremental induction has been developed at the bit level [219, 220], showing significant performance advantages compared to the original incremental induction algorithm. Exploring a similar approach both, for word-level hardware verification as well as distributed protocol verification, is an interesting direction that can lead to improving the scalability of verification to increasingly complex practical systems.

Inferring Intermediate Specifications: Assuming a hierarchically specified protocol with a sufficient number of levels, Chapter 6 showed that it is possible to obtain a fully-automatic procedure for hierarchical invariant generation based on symmetry and range boosting. Of course, this begs the question of *what if there is no hierarchy or insufficient levels*. The original Paxos specification is composed of a two-level hierarchy $Paxos \prec Voting$. While introducing *ImplicitPaxos* and *SimplePaxos* to get the four-level Paxos hierarchy was quite easy, these intermediate levels were still added manually. It is appealing to explore CEGAR-based techniques to automatically identify these abstract intermediate specifications whenever needed to overcome complexity. Specifically, investigating how to leverage clause learning feedback from incomplete verification runs to identify bottlenecks in proof inference and utilizing this information to automatically abstract away irrelevant details from the low-level protocol can help in making the complete procedure automatic end-to-end.

Theoretical Exploration: As detailed in Section 5.7, IC3PO includes a finite convergence procedure that combines notions of symmetry saturation and multi-dimensional mathematical induction. Empirically, for all symmetric protocols we tested in Section 5.10, the unbounded inductiveness check passes, which suggests that the finite convergence procedure is correctly finding the cutoff size for deriving the inductive invariant. However, we do not have a formal proof of this procedure. Furthermore, for some protocols, the inferred quantified invariant is not in the EPR fragment of first-order logic, and the unbounded check may not succeed using current SMT solving techniques, though laborious manual proofs using interactive theorem provers can establish their correctness in the unbounded setting. It is interesting to develop a better theoretical understanding of the strengths and limitations of such finite analysis, particularly for the class of unbounded protocols for which it is applicable. A more speculative related problem is the possibility of identifying a new decidable fragment of first-order logic, namely the class that is based on symmetric sorts which does not adhere to the EPR fragment restrictions by allowing function symbols and arbitrary quantifier alternations. IC3PO's finite convergence handles such cases and the fact that they also pass the unbounded check is suggestive of decidability and proving this would be an interesting theoretical outcome.

Exploring Different Applications: The ideas of structural regularity and data abstraction are not unique to distributed protocols or hardware systems, and we expect the ideas developed in this research to be applicable to other domains. For example, many algorithms in software-defined networking present symmetry between nodes and are also in need of formal verification. We expect that the ideas presented in this dissertation are also applicable to other fields, especially the ones where finding a inductive invariant is currently a hurdle. Specifically, employing similar techniques for verifying smart contracts and consensus algorithms used in blockchain applications can be of particular interest to offer provable security and correctness guarantees.

APPENDIX A

Additional Material on Verifying Distributed Protocols

We include additional/supplementary material in the appendices, as follows:

A.1: *IC3PO Pseudo Code (detailed)*

- Presents the detailed pseudo code of IC3PO and *SymIC3*

A.2: *Proof of Correctness*

- Provides a correctness proof for symmetry boosting during incremental induction (Section 5.5), and a correctness proof for quantifier inference (Section 5.6)

A.3: *Finite Instance Sizes used in the Experiments*

- Lists down the instance sizes for IC3PO and I4 for each protocol in the evaluation (Section 5.10)

A.1 IC3PO Pseudo Code

This section presents the detailed pseudo code of IC3PO and *SymIC3*.

Algorithm 7 ***IC3 for Proving Protocol Properties***

```
1  procedure IC3PO($\mathcal{P}$, $\sigma_0$)        ▷ $\mathcal{P} \triangleq [S, R, Init, T, P]$, and $\sigma_0$ is the initial base size
2      $reuse \leftarrow \{\}$
3      $\sigma \leftarrow \sigma_0$
4      $Inv, Cex \leftarrow$ SymIC3($\hat{\mathcal{P}}$, $reuse$)        ▷ run symmetric incremental induction on $\hat{\mathcal{P}} \triangleq \mathcal{P}(\sigma)$
5      if $Cex$ is not empty then        ▷ counterexample found
6          return Violated, $Cex$        ▷ property is violated
7      else        ▷ property proved for the finite protocol instance $\hat{\mathcal{P}}$
8          for each $\mathtt{s_i} \in S$ do
9              if not IsInductiveInvariantFinite($Inv$, $\mathcal{P}(\sigma^{+}[\mathtt{s_i}])$) then
10                 $reuse \leftarrow \{\, \Phi \mid \Phi \in Inv$ and $Init \rightarrow \Phi$ and $Init \wedge T \rightarrow \Phi'$ in $\mathcal{P}(\sigma^{+}[\mathtt{s_i}])\, \}$
11                 $\sigma \leftarrow \sigma^{+}[\mathtt{s_i}]$        ▷ failed convergence checks for sort $\mathtt{s_i}$, increase instance size
12                 go to Line 4        ▷ re-run SymIC3 with the increased size
13         return Safe, $Inv$        ▷ property is proved safe with proof certificate $Inv$
```

Algorithm 7 presents the detailed pseudo code of IC3PO. Let $\sigma : S \rightarrow \mathbb{N}$ be a function that maps each sort $\mathtt{s_i} \in S$ to a sort size $|\mathtt{s_i}|$. Given a protocol specification $\mathcal{P}$ and an initial base size σ_0, IC3PO invokes *SymIC3* on the finite protocol instance $\hat{\mathcal{P}} \triangleq \mathcal{P}(\sigma)$, where σ is initialized to σ_0 (lines 2-4). Upon termination, *SymIC3* either a) produces a quantified inductive invariant Inv that proves the property for $\hat{\mathcal{P}}$, or b) a counterexample trace Cex that serves as a finite witness to its violation in both $\hat{\mathcal{P}}$ and the unbounded protocol $\mathcal{P}$ (lines 4-6). If the property holds for $\hat{\mathcal{P}}$, IC3PO performs finite convergence checks (Section 5.7) to check whether or not the invariant extends beyond $\hat{\mathcal{P}}$ (lines 8-12), by checking whether or not Inv is an inductive invariant for the larger finite instance $\hat{\mathcal{P}}^{\mathtt{i}} \triangleq \mathcal{P}(\sigma^{+}[\mathtt{s_i}])$ for each $\mathtt{s_i} \in S$, where $\sigma^{+}[\mathtt{s_i}] \triangleq [\sigma \;\; \mathrm{EXCEPT}\; ![\mathtt{s_i}] = \sigma(\mathtt{s_i}) + 1]$. If all these checks pass, IC3PO emits the unbounded invariant Inv, that holds for the unbounded $\mathcal{P}$ and is a proof certificate for the safety property (line 13). Otherwise, it re-starts *SymIC3* on a finite instance with an increased size $\sigma^{+}[\mathtt{s_i}]$ (lines 11-12), while seeding in all the strengthening assertions in Inv that are safe to learn in the first frame for the new *SymIC3* iteration (line 10).

Algorithm 8 describes the symmetric incremental induction algorithm. The procedure first checks whether the property can be trivially violated (lines 16-19), and if not, starts recursively deriving and blocking counterexamples-to-induction (CTI) from the topmost frame (lines 21-32). Given a solver model $\mathtt{m}$, a state cube is derived as a single state represented as a cube, i.e., a conjunction of literals assigning each state variable with a value based on its assignment in $\mathtt{m}$ (lines

Algorithm 8 *Symmetric Incremental Induction*

```
14 procedure SymIC3($\hat{\mathcal{P}}$, $reuse$)    ▷ $\hat{\mathcal{P}} \triangleq [S, R, \hat{Init}, \hat{T}, \hat{P}]$
                                                     ▷ $reuse$ is a set of seed assertions that are safe to learn in the frame $F_1$
15     $F \leftarrow \emptyset$, $Cex \leftarrow \emptyset$    ▷ $\hat{\mathcal{P}}$, $F$, $Cex$ are global data structures
16     if SAT ? [ $\hat{Init} \wedge \neg\hat{P}$ ]: model m then    ▷ initial states check
17         $state \leftarrow$ StateAsCube(m)    ▷ get a single state from model m, in cube form
18         $Cex.extend(state)$    ▷ property is trivially violated
19         return $\emptyset$, $Cex$    ▷ return the counterexample
20     $F.extend(\hat{Init})$    ▷ setup the initial frame
21     while $\top$ do
22         $N \leftarrow F.size() - 1$
23         if SAT ? [ $F_N \wedge \hat{T} \wedge \neg\hat{P}'$ ]: model m then    ▷ check the topmost frame for CTI
24             $state \leftarrow$ StateAsCube(m)    ▷ found a CTI
25             if SymRecBlockCube($state$, $N$) then    ▷ try recursively blocking the CTI
26                 return $\emptyset$, $Cex$    ▷ failed to block CTI, return the counterexample
27         else    ▷ no CTI in the topmost frame
28             $F.extend(\hat{P})$    ▷ add a new frame
29             if $N = 0$ then    ▷ add reusable seed assertions to the frame $F_1$
30                 $F[1].add(reuse)$
31             if ForwardPropagate() then    ▷ propagate inductive assertions forward
32                 return $F_{converged}$, $\emptyset$    ▷ frames converged, return $F_{converged}$ as the inductive invariant

33 procedure SymRecBlockCube($cti$, $i$)    ▷ $cti$ can reach $\neg\hat{P}$ in $F.size() - i$ steps
34     $Cex.extend(cti)$    ▷ add the CTI to the counterexample
35     if $i = 0$ then    ▷ check if reached the initial states
36         return $\top$    ▷ reached initial states, property is violated
37     if SAT ? [ $F_{i-1} \wedge \hat{T} \wedge cti'$ ]: model m then
                                                     ▷ check if $cti$ is reachable from previous frame
38         $state \leftarrow$ StateAsCube(m)
                                                     ▷ $state$ is the new CTI reachable to $\neg\hat{P}$ in $(F.size() - i) + 1$ steps
39         return SymRecBlockCube($state$, $i - 1$)    ▷ try blocking the new CTI
40     else    ▷ $cti$ is unreachable from the previous frame
41         $uc' \leftarrow$ MinimalUnsatCore($F_{i-1} \wedge \hat{T}$, $cti'$)    ▷ get MUS from UNSAT query
42         $\varphi \leftarrow \neg uc$    ▷ negate $uc$ to get the quantifier-free clause
43         $\Phi \leftarrow$ SymBoost$\forall\exists(\varphi)$    ▷ symmetry boosting with quantifier inference
44         $\Phi \leftarrow$ AntecedentReduction($\Phi$, $i$)    ▷ antecedent reduction (optional)
45         $\Phi \leftarrow$ PushOutExists($\Phi$, $i$)    ▷ pushing out existential (optional)
46         Learn($\Phi$, $F_i$)    ▷ learn $\Phi$ in frame $i$
47         return $\bot$
```

17, 24, 38). Lines 29-30 add the seed assertions in the given $reuse$ set to the first frame F_1. ***SymIC3*** differs from the standard IC3 algorithm majorly in symmetry-aware quantified learning (line 43) and simple enhancements (lines 44-45).

Algorithm 9 *Symmetry Boosting with Quantifier Inference*

```
48 procedure SymBoost∀∃($\varphi$)                                    ▷ $\varphi$ is the quantifier-free clause
49     $V_\forall \leftarrow \{\}$, $V_\exists \leftarrow \{\}$          ▷ a set of universally/existential quantified variables
50     body ← $\varphi$                                  ▷ starting with $\varphi$, body is recursively generated
                                                      ▷ $V_\forall$, $V_\exists$ and body are global data structures
51     for each sort s that appears in clause $\varphi$ do
52         $\pi(\varphi, s)$ ← PARTITIONDISTRIBUTION($\varphi$, s)
                              ▷ create a partition on constants in s based on their occurrence in $\varphi$
53         if $\#(\varphi, s) < |s|$ then
54             ($V_\forall$, $V_\exists$, body) ← INFER∀($\varphi$, $\pi(\varphi, s)$)        ▷ infer ∀ for sort s, refer §5.6.1.A
55         else if $|\pi(\varphi, s)| = 1$ then                ▷ partition $\pi(\varphi, s)$ contains a single cell
56             ($V_\forall$, $V_\exists$, body) ← INFER∃($\varphi$, $\pi(\varphi, s)$)      ▷ infer ∃ for sort s, refer §5.6.1.B.I
57         else if all but a few scenario then          ▷ partition $\pi(\varphi, s)$ contains multiple cells
58             ($V_\forall$, $V_\exists$, body) ← INFER∀∃($\varphi$, $\pi(\varphi, s)$)    ▷ infer ∀∃ for sort s, refer §5.6.1.B.II
59         else
60             < never occurred >
                     ▷ infer ∀ by default (may not be compact, though correct for the current instance)
61             ($V_\forall$, $V_\exists$, body) ← INFER∀($\varphi$, $\pi(\varphi, s)$)
62     $\Phi \leftarrow \forall V_\forall .\ \exists V_\exists .$ body        ▷ stitch quantifiers for different sorts as $\forall_{\ldots}\ \exists_{\ldots}$ < body >
63     return $\Phi$                          ▷ $\Phi$ is the quantified predicate to learn in a SymIC3 frame
```

The core of the *SymIC3* algorithm is the *SymBoost*∀∃ algorithm, presented in Algorithm 9. *SymBoost*∀∃ is a simple and extendable procedure to perform symmetry boosting and quantifier inference, as explained in detail in Sections 5.5 and 5.6. Starting from a given quantifier-free clause φ, the algorithm constructs a symmetrically-boosted quantified predicate Φ (line 63) by iteratively inferring quantifiers for each sort s (lines 51-61), and stitching them together (line 62). The algorithm maintains a set of universal and existential variables (line 49) and a *body* (line 50), that are iteratively modified based on the quantifier inference for each sort. For each sort s, the algorithm first generates $\pi(\varphi, s)$ (line 52) based on how constants in sort s appear in the literals of φ (whether identically or not). The next step is to infer quantifiers using $\#(\varphi, s)$ and $\pi(\varphi, s)$ (lines 53-61): a) infer universal quantifiers when $\#(\varphi, s) < |s|$, b) otherwise if all constants of s appear in φ identically, infer existential quantifier, c) otherwise if *all but a few scenario*, infer ∀∃ based on the partitioning of constants in $\pi(\varphi, s)$, and d) otherwise, infer ∀ by default (this case has not occurred). Changing the iteration order in line 51 doesn't result in any difference and is ensured during the recursive building of the *body*. At the end, a single quantified predicate Φ is derived by stitching together the quantified variables in $V_\forall$ and $V_\exists$ with the *body* as $\forall_{\ldots}\ \exists_{\ldots}\ <$ *body* $>$ (line 62).

A.2 Proof of Correctness

A.2.1 Correctness Proof for Symmetric Incremental Induction

This section provides a correctness proof for symmetry boosting during incremental induction (Section 5.5).

Like the invariance of $\hat{Init}$, $\hat{T}$, and $\hat{P}$ under any permutation $\gamma \in G$ (refer (5.2)), the logical orbit of a clause φ is also invariant under such permutations, i.e.,

$$\left[\varphi^{L(G)}\right]^{\gamma} \leftrightarrow \varphi^{L(G)}$$

Lemma 4. *For any SymIC3 frame F_i, $F_i^{\gamma} \equiv F_i$ for any $\gamma \in G$.*

Proof. Recall that $\hat{Init}^{\gamma} \equiv \hat{Init}$ and $\hat{P}^{\gamma} \equiv \hat{P}$. The condition $F_i^{\gamma} \equiv F_i$ is trivially true for $i = 0$ since $F_0 = \hat{Init}$. When $i > 0$, the condition is true during frame initialization since each frame is initialized to $\hat{P}$. When blocking a cube $\neg\varphi$ in F_i, incremental induction with symmetry boosting refines F_i with the complete logical orbit $\varphi^{L(G)}$ of φ. Since $\left[\varphi^{L(G)}\right]^{\gamma} \equiv \varphi^{L(G)}$, the logical invariance of F_i under γ, continues to be preserved in all backward reachability updates. □

The following theorem establishes the correctness of symmetry boosting in incremental induction.

Theorem A.2.1. *If a quantifier-free cube $\neg\varphi$ is unreachable from frame F_{i-1}, i.e., $F_{i-1} \wedge \hat{T} \wedge \neg[\varphi]'$ is unsatisfiable, then $F_{i-1} \wedge \hat{T} \wedge \neg[\varphi^{L(G)}]'$ is also unsatisfiable.*

Proof. Let $Q \triangleq F_{i-1} \wedge \hat{T} \wedge \neg[\varphi]'$ and assume that Q is unsatisfiable. Consider any permutation $\gamma \in G$ and the corresponding permuted formula $Q^{\gamma} \triangleq F_{i-1}^{\gamma} \wedge \hat{T}^{\gamma} \wedge \neg\left[\varphi^{\gamma}\right]'$. Since permuting the sort constants simply re-arranges the protocol's state variables in a formula without affecting its satisfiability, Q and Q^{γ} must be equisatisfiable, and hence Q^{γ} is unsatisfiable.

Noting that $\hat{T}$ and F_{i-1} are invariant under $\gamma \in G$ (from (5.2) and Lemma 4), we obtain $Q^{\gamma} = F_{i-1} \wedge \hat{T} \wedge \neg\left[\varphi^{\gamma}\right]'$ proving that if cube $\neg\varphi$ is unreachable from frame F_{i-1}, then its image under any $\gamma \in G$ is also unreachable. Therefore, $F_{i-1} \wedge \hat{T} \wedge \neg[\varphi^{L(G)}]'$ is unsatisfiable. □

A.2.2 Correctness Proof for Quantifier Inference

This section provides a correctness proof sketch for quantifier inference (Section 5.6).

Theorem A.2.2. *Given a finite instance $\hat{\mathcal{P}}$, let φ be such that $0 < \#(\varphi, \mathtt{s}) < |\mathtt{s}|$ for some sort $\mathtt{s} \in S$. Let $\Phi(\mathtt{s})$ be the quantified predicate obtained by applying SymIC3's quantifier inference for* $\mathtt{s}$. *$\Phi(\mathtt{s})$ is logically equivalent to $\varphi^{L(Sym(\mathtt{s}))}$.*

Proof. Let γ be any permutation in $Sym(\mathtt{s})$, and let $n \triangleq \#(\varphi, \mathtt{s})$. Let $\widehat{\varphi}$ be the clause obtained by replacing in φ each constant $\mathtt{c_i} \in \mathtt{s}$ by a corresponding variable V_i of sort $\mathtt{s}$.

Let $A \triangleq [\,(V_1 = \mathtt{c_1}) \wedge \cdots \wedge (V_n = \mathtt{c_n})\,] \rightarrow \widehat{\varphi}$. By the transitivity of equality, $A \equiv \varphi$. Let $B \triangleq \bigwedge_{\gamma \,\in\, Sym(\mathtt{s})} A^{\gamma}$. Since $A \equiv \varphi$, therefore, $B \equiv \varphi^{L(Sym(\mathtt{s}))}$, and can be re-written as:

$$B = \bigwedge_{\gamma \,\in\, Sym(\mathtt{s})} \left(\,[\,(V_1 = \mathtt{c_1}) \wedge \cdots \wedge (V_n = \mathtt{c_n})\,] \rightarrow \widehat{\varphi}\,\right)^{\gamma} \tag{A.1}$$

$$= \bigwedge_{\gamma \,\in\, Sym(\mathtt{s})} [\,(V_1 = \mathtt{c_1}) \wedge \cdots \wedge (V_n = \mathtt{c_n})\,]^{\gamma} \rightarrow \widehat{\varphi} \tag{A.2}$$

$$= \forall\ V_1 \ldots V_n.\ (\text{distinct } V_1 \ldots V_n) \rightarrow \widehat{\varphi} \tag{A.3}$$

$$= \Phi(\mathtt{s}) \tag{A.4}$$

(A.1) & (A.2) are equal since $\widehat{\varphi}$ does not contain any constant of sort $\mathtt{s}$, and hence $[\widehat{\varphi}]^{\gamma} \equiv \widehat{\varphi}$. (A.2) & (A.3) are equal since the antecedents in (A.2) cover all possible assignments of variables $(V_1, \ldots, V_n)$ to n distinct constants of sort $\mathtt{s}$. There are total $\binom{|\mathtt{s}|}{n} \times n!$ possible assignments of the variables in (A.3) to n distinct constants of sort $\mathtt{s}$, one each corresponding to the $\binom{|\mathtt{s}|}{n} \times n!$ permutations in $Sym(\mathtt{s})$ that yield a logically-distinct antecedent in (A.2). (A.3) & (A.4) are equal since given $\#(\varphi, \mathtt{s}) < |\mathtt{s}|$.

Since $B \equiv \varphi^{L(Sym(\mathtt{s}))}$, therefore $\Phi(\mathtt{s}) \equiv \varphi^{L(Sym(\mathtt{s}))}$. □

Theorem A.2.3. *Given a finite instance $\hat{\mathcal{P}}$, let φ be such that all constants of a sort $\mathtt{s} \in S$ appear identically in the literals of φ. Let $\Phi(\mathtt{s})$ be the quantified predicate obtained by applying SymIC3's quantifier inference for* $\mathtt{s}$. *$\Phi(\mathtt{s})$ is logically equivalent to $\varphi^{L(Sym(\mathtt{s}))}$.*

Proof. Let γ be any permutation in $Sym(\mathtt{s})$. Since given all constants in sort $\mathtt{s}$ appear identically in the literals of φ, therefore $\pi(\varphi, \mathtt{s})$ consists of a single cell, and any permutation $\gamma \in Sym(\mathtt{s})$ does not result in a new logically-distinct clause, i.e., $\varphi^{\gamma} \equiv \varphi$. As a result, $\varphi^{L(Sym(\mathtt{s}))} \equiv \varphi$.

Without loss of generality, φ can be written as:

$$\varphi = \varphi_{others} \vee \bigvee_{\mathtt{c_i} \,\in\, \mathtt{s}} \varphi_{\mathtt{s}}(\mathtt{c_i}) \tag{A.5}$$

where φ_{others} is the disjunction of literals in φ that do not contain any constant of sort $\mathtt{s}$, and $\varphi_{\mathtt{s}}(\mathtt{c_i})$ is the disjunction of literals in φ that contain a constant $\mathtt{c_i} \in \mathtt{s}$. Note that φ_{others} can be $\bot$.

Let $\widehat{\varphi_{\mathtt{s}}}$ be the clause obtained by replacing in $\varphi_{\mathtt{s}}(\mathtt{c_i})$ each constant $\mathtt{c_i} \in \mathtt{s}$ by a variable V of sort $\mathtt{s}$. Note that since all constants of sort $\mathtt{s}$ appear identically in the literals of φ, therefore $\widehat{\varphi_{\mathtt{s}}}$ is the same for each $\mathtt{c_i} \in \mathtt{s}$. The clause φ can therefore be re-written as:

$$\varphi = \varphi_{others} \vee \bigvee_{\mathtt{c_i} \in \mathtt{s}} (V = \mathtt{c_i}) \rightarrow \widehat{\varphi_{\mathtt{s}}} \tag{A.6}$$

$$= \varphi_{others} \vee \exists\, V.\ \widehat{\varphi_{\mathtt{s}}} \tag{A.7}$$

$$= \Phi(\mathtt{s}) \tag{A.8}$$

(A.5) & (A.6) are equal due to the transitivity of equality. (A.6) & (A.7) are equal since expanding the existential quantifier as a disjunction over all possible assignments of the variable V gives the expression in (A.6). (A.7) & (A.8) are equal since $\#(\varphi, \mathtt{s}) = |\mathtt{s}|$ and $|\pi(\varphi, \mathtt{s}) = 1|$, and hence *SymIC3* infers $\Phi(\mathtt{s})$ as (A.7). Since $\varphi \equiv \varphi^{L(Sym(\mathtt{s}))}$, therefore $\Phi(\mathtt{s}) \equiv \varphi^{L(Sym(\mathtt{s}))}$. □

A.3 Finite Instance Sizes used in Experiments

Table B.1 lists down the initial base instance sizes used for IC3PO runs in the evaluation (Section 5.10) for each protocol. The table also includes the final *cutoff* instance sizes reached, where the corresponding *Inv* generalizes/saturates to be an inductive proof for any size. Note again that IC3PO updates the instance sizes automatically, as described in Section 5.7.

Table B.2 lists down the instance sizes used for I4 runs in the evaluation (Section 5.10) for each protocol.

Protocol	Finite instance sizes used for IC3PO
tla-consensus	$\texttt{value} = 2$
tla-tcommit	$\texttt{resource-manager} = 2$
i4-lock-server	$\texttt{client} = 2,\ \texttt{server} = 1$
ex-quorum-leader-election E	$\texttt{node} = 2 \mapsto 3,\ \texttt{nset} = 2$
pyv-toy-consensus-forall E	$\texttt{node} = 2 \mapsto 3,\ \texttt{quorum} = 1 \mapsto 3,\ \texttt{value} = 2$
tla-simple $\circlearrowright$ E	$\texttt{node} = 2,\ \texttt{pcstate} = 3,\ \texttt{value} = 2 \mapsto 3$
ex-lockserv-automaton	$\texttt{node} = 2$
tla-simpleregular $\circlearrowright$ E	$\texttt{node} = 2,\ \texttt{pcstate} = 4,\ \texttt{value} = 2 \mapsto 3$
pyv-sharded-kv	$\texttt{key} = 2,\ \texttt{node} = 2,\ \texttt{value} = 2$
pyv-lockserv	$\texttt{node} = 2$
tla-twophase	$\texttt{resource-manager} = 2$
i4-learning-switch	$\texttt{node} = 2 \mapsto 3,\ \texttt{packet} = 1$
ex-simple-decentralized-lock	$\texttt{node} = 2 \mapsto 4$
i4-two-phase-commit	$\texttt{node} = 4$
pyv-consensus-wo-decide E	$\texttt{node} = 2 \mapsto 3,\ \texttt{quorum} = 1 \mapsto 3$
pyv-consensus-forall E	$\texttt{node} = 2 \mapsto 3,\ \texttt{quorum} = 1 \mapsto 3,\ \texttt{value} = 2$
pyv-learning-switch E	$\texttt{node} = 2 \mapsto 4$
i4-chord-ring-maintenance $\circlearrowright$ E	$\texttt{node} = 3 \mapsto 5$
pyv-sharded-kv-no-lost-keys E	$\texttt{key} = 2,\ \texttt{node} = 2,\ \texttt{value} = 2$
ex-naive-consensus E	$\texttt{node} = 3,\ \texttt{quorum} = 3,\ \texttt{value} = 3$
pyv-client-server-ae E	$\texttt{node} = 2,\ \texttt{request} = 2 \mapsto 3,\ \texttt{response} = 2$
ex-simple-election E	$\texttt{acceptor} = 2 \mapsto 3,\ \texttt{proposer} = 2,\ \texttt{quorum} = 1 \mapsto 3$
pyv-toy-consensus-epr E	$\texttt{node} = 2 \mapsto 3,\ \texttt{quorum} = 1 \mapsto 3,\ \texttt{value} = 2$
ex-toy-consensus E	$\texttt{node} = 2 \mapsto 3,\ \texttt{quorum} = 1 \mapsto 3,\ \texttt{value} = 2$
pyv-client-server-db-ae E	$\texttt{db-request-id} = 2 \mapsto 3,\ \texttt{node} = 2,\ \texttt{request} = 2 \mapsto 3,\ \texttt{response} = 2$
pyv-hybrid-reliable-broadcast E	$\texttt{node} = 2 \mapsto 3,\ \texttt{quorum-a} = 2 \mapsto 3,\ \texttt{quorum-b} = 2$
pyv-firewall E	$\texttt{node} = 2 \mapsto 3$
ex-majorityset-leader-election E	$\texttt{node} = 2 \mapsto 3,\ \texttt{nodeset} = 2 \mapsto 3$
pyv-consensus-epr E	$\texttt{node} = 2 \mapsto 3,\ \texttt{quorum} = 1 \mapsto 3,\ \texttt{value} = 2$
ex-distributed-lock-abstract $<$	$\texttt{epoch} = \infty,\ \texttt{node} = 2$
ex-decentralized-lock $<$	$\texttt{node} = 2,\ \texttt{time} = \infty$
ex-distributed-lock-maxheld $<$	$\texttt{epoch} = \infty,\ \texttt{node} = 2$
pyv-ticket $<$	$\texttt{thread} = 2 \mapsto 3,\ \texttt{ticket} = \infty$
i4-database-chain-replication E $<$	$\texttt{key} = 1,\ \texttt{node} = 2,\ \texttt{operation} = 2 \mapsto 3,\ \texttt{transaction} = \infty$
ex-decentralized-lock-abstract $<$	$\texttt{node} = 2 \mapsto 4,\ \texttt{time} = \infty$
i4-distributed-lock $<$	$\texttt{epoch} = \infty,\ \texttt{node} = 2$
ex-ring-not-dead $\circlearrowright$ E $<$	$\texttt{node} = 3$
ex-ring $\circlearrowright$ $<$	$\texttt{node} = 3$
ex-ring-id-not-dead-limited $\circlearrowright$ E $<$	$\texttt{id} = 3,\ \texttt{node} = 3$
pyv-ring-id-not-dead $\circlearrowright$ E $<$	$\texttt{id} = \infty,\ \texttt{node} = 3$
pyv-ring-id $\circlearrowright$ $<$	$\texttt{id} = \infty,\ \texttt{node} = 3$
i4-leader-election-in-ring $\circlearrowright$ $<$	$\texttt{id} = \infty,\ \texttt{node} = 3$

Table A.1: Finite instance sizes used for IC3PO

$\texttt{s} = x$ denotes sort $\texttt{s}$ has both initial base size and final cutoff size x
$\texttt{s} = x \mapsto y$ denotes sort $\texttt{s}$ has initial size x and final cutoff size y (incrementally increased by IC3PO automatically)
$\texttt{s} = \infty$ denote the totally-ordered sort $\texttt{s}$ is left unbounded
$\circlearrowright$ indicates protocol has a ring topology, $<$ indicates protocol has an ordered domain
E indicates the protocol description has $\exists$

Protocol	Finite instance sizes used for I4
tla-consensus	$\texttt{value} = 2$
tla-tcommit	$\texttt{resource-manager} = 2$
i4-lock-server	$\texttt{client} = 2,\ \texttt{server} = 1$
ex-quorum-leader-election E	$\texttt{node} = 3,\ \texttt{nset} = 3$
pyv-toy-consensus-forall E	$\texttt{node} = 3,\ \texttt{quorum} = 3,\ \texttt{value} = 2$
tla-simple ↻ E	$\texttt{node} = 3,\ \texttt{pcstate} = 3,\ \texttt{value} = 3$
ex-lockserv-automaton	$\texttt{node} = 2$
tla-simpleregular ↻ E	$\texttt{node} = 3,\ \texttt{pcstate} = 4,\ \texttt{value} = 3$
pyv-sharded-kv	$\texttt{key} = 2,\ \texttt{node} = 2,\ \texttt{value} = 2$
pyv-lockserv	$\texttt{node} = 2$
tla-twophase	$\texttt{resource-manager} = 3$
i4-learning-switch	$\texttt{node} = 3,\ \texttt{packet} = 2$
ex-simple-decentralized-lock	$\texttt{node} = 4$
i4-two-phase-commit	$\texttt{node} = 5$
pyv-consensus-wo-decide E	$\texttt{node} = 3,\ \texttt{quorum} = 3$
pyv-consensus-forall E	$\texttt{node} = 3,\ \texttt{quorum} = 3,\ \texttt{value} = 2$
pyv-learning-switch E	$\texttt{node} = 4$
i4-chord-ring-maintenance ↻ E	$\texttt{node} = 4$
pyv-sharded-kv-no-lost-keys E	$\texttt{key} = 3,\ \texttt{node} = 3,\ \texttt{value} = 3$
ex-naive-consensus E	$\texttt{node} = 3,\ \texttt{quorum} = 3,\ \texttt{value} = 3$
pyv-client-server-ae E	$\texttt{node} = 3,\ \texttt{request} = 3,\ \texttt{response} = 3$
ex-simple-election E	$\texttt{acceptor} = 3,\ \texttt{proposer} = 2,\ \texttt{quorum} = 3$
pyv-toy-consensus-epr E	$\texttt{node} = 3,\ \texttt{quorum} = 3,\ \texttt{value} = 2$
ex-toy-consensus E	$\texttt{node} = 3,\ \texttt{quorum} = 3,\ \texttt{value} = 2$
pyv-client-server-db-ae E	$\texttt{db-request-id} = 3,\ \texttt{node} = 3,\ \texttt{request} = 3,\ \texttt{response} = 3$
pyv-hybrid-reliable-broadcast E	$\texttt{node} = 3,\ \texttt{quorum-a} = 3,\ \texttt{quorum-b} = 3$
pyv-firewall E	$\texttt{node} = 3$
ex-majorityset-leader-election E	$\texttt{node} = 3,\ \texttt{nodeset} = 3$
pyv-consensus-epr E	$\texttt{node} = 3,\ \texttt{quorum} = 3,\ \texttt{value} = 2$
ex-distributed-lock-abstract $<$	$\texttt{epoch} = 4,\ \texttt{node} = 2$
ex-decentralized-lock $<$	$\texttt{node} = 2,\ \texttt{time} = 4$
ex-distributed-lock-maxheld $<$	$\texttt{epoch} = 4,\ \texttt{node} = 2$
pyv-ticket $<$	$\texttt{thread} = 3,\ \texttt{ticket} = 5$
i4-database-chain-replication $E <$	$\texttt{key} = 1,\ \texttt{node} = 2,\ \texttt{operation} = 3,\ \texttt{transaction} = 3$
ex-decentralized-lock-abstract $<$	$\texttt{node} = 4,\ \texttt{time} = 4$
i4-distributed-lock $<$	$\texttt{epoch} = 4,\ \texttt{node} = 2$
ex-ring-not-dead ↻ $E <$	$\texttt{node} = 3$
ex-ring ↻ $<$	$\texttt{node} = 3$
ex-ring-id-not-dead-limited ↻ $E <$	$\texttt{id} = 3,\ \texttt{node} = 3$
pyv-ring-id-not-dead ↻ $E <$	$\texttt{id} = 4,\ \texttt{node} = 3$
pyv-ring-id ↻ $<$	$\texttt{id} = 4,\ \texttt{node} = 3$
i4-leader-election-in-ring ↻ $<$	$\texttt{id} = 4,\ \texttt{node} = 3$

Table A.2: Finite instance sizes used for I4
↻ indicates protocol has a ring topology, $<$ indicates protocol has an ordered domain
E indicates the protocol description has $\exists$

APPENDIX B

Additional Material on Verifying Paxos

We include additional/supplementary material in the appendices, as follows:

B.1: *Finite instance sizes used in the experiments*

- Lists down the instance sizes for IC3PO and I4 for each protocol in the evaluation (Section 6.7)

B.2: *TLA+ description for SimplePaxos and ImplicitPaxos*

- Presents full TLA+ descriptions of *SimplePaxos* and *ImplicitPaxos*

B.1 Finite Instance Sizes used in the Experiments

B.1.1 Finite Instance Sizes for IC3PO

Table B.1 lists down the initial base instance sizes used for IC3PO runs in the evaluation (Section 6.7) for each protocol. The table also includes the final *cutoff* instance sizes reached, where the corresponding inductive invariant generalizes to be an inductive proof for *any* size. Note again that IC3PO checks for finite convergence and updates the instance sizes automatically, as detailed in Section 5.7.

Protocol		Finite instance sizes used for IC3PO
$\texttt{EPR}$	epr-paxos	$\texttt{value} = 2,\ \texttt{node} = 3,\ \texttt{quorum} = 3,\ \texttt{round} = 4$
	epr-flexible_paxos	$\texttt{value} = 2,\ \texttt{node} = 3,\ \texttt{quorum1} = 3,\ \texttt{quorum2} = 3,\ \texttt{round} = 4$
	epr-multi_paxos	$\texttt{value} = 2,\ \texttt{node} = 3,\ \texttt{quorum} = 3,\ \texttt{round} = 4,\ \texttt{inst} = 2,\ \texttt{votemap} = 2$
$\texttt{ORIGINAL}$	*Voting*	$\texttt{value} = 2,\ \texttt{acceptor} = 3,\ \texttt{quorum} = 3,\ \texttt{ballot} = 4$
	SimplePaxos	$\texttt{value} = 2,\ \texttt{acceptor} = 3,\ \texttt{quorum} = 3,\ \texttt{ballot} = 4$
	ImplicitPaxos	$\texttt{value} = 2,\ \texttt{acceptor} = 3,\ \texttt{quorum} = 3,\ \texttt{ballot} = 4 \mapsto 5$
	Paxos	$\texttt{value} = 2,\ \texttt{acceptor} = 3,\ \texttt{quorum} = 3,\ \texttt{ballot} = 4$
	MultiPaxos	$\texttt{value} = 2,\ \texttt{acceptor} = 3,\ \texttt{quorum} = 3,\ \texttt{ballot} = 4,\ \texttt{instances} = 2$
	FlexiblePaxos	$\texttt{value} = 2,\ \texttt{acceptor} = 3,\ \texttt{quorum1} = 3,\ \texttt{quorum2} = 3,\ \texttt{ballot} = 4$

Table B.1: Finite instance sizes used for IC3PO

$\texttt{s} = x$ denotes sort $\texttt{s}$ has both initial base size and final cutoff size x

$\texttt{s} = x \mapsto y$ denotes sort $\texttt{s}$ has initial size x and final cutoff size y (only happens for *ImplicitPaxos*)

B.1.2 Finite Instance Sizes for I4

Table B.2 lists down the instance sizes used for I4 runs in the evaluation (Section 6.7) for each protocol.

Protocol		Finite instance sizes used for I4
$\texttt{EPR}$	epr-paxos	$\texttt{value} = 2,\ \texttt{node} = 3,\ \texttt{quorum} = 3,\ \texttt{round} = 4$
	epr-flexible_paxos	$\texttt{value} = 2,\ \texttt{node} = 3,\ \texttt{quorum1} = 3,\ \texttt{quorum2} = 3,\ \texttt{round} = 4$
	epr-multi_paxos	$\texttt{value} = 2,\ \texttt{node} = 3,\ \texttt{quorum} = 3,\ \texttt{round} = 4,\ \texttt{inst} = 2,\ \texttt{votemap} = 2$
$\texttt{ORIGINAL}$	*Voting*	$\texttt{value} = 2,\ \texttt{acceptor} = 3,\ \texttt{quorum} = 3,\ \texttt{ballot} = 4$
	SimplePaxos	$\texttt{value} = 2,\ \texttt{acceptor} = 3,\ \texttt{quorum} = 3,\ \texttt{ballot} = 4$
	ImplicitPaxos	$\texttt{value} = 2,\ \texttt{acceptor} = 3,\ \texttt{quorum} = 3,\ \texttt{ballot} = 5$
	Paxos	$\texttt{value} = 2,\ \texttt{acceptor} = 3,\ \texttt{quorum} = 3,\ \texttt{ballot} = 4$
	MultiPaxos	$\texttt{value} = 2,\ \texttt{acceptor} = 3,\ \texttt{quorum} = 3,\ \texttt{ballot} = 4,\ \texttt{instances} = 2$
	FlexiblePaxos	$\texttt{value} = 2,\ \texttt{acceptor} = 3,\ \texttt{quorum1} = 3,\ \texttt{quorum2} = 3,\ \texttt{ballot} = 4$

Table B.2: Finite instance sizes used for I4

B.2 TLA+ description for *SimplePaxos* and *ImplicitPaxos*

This section presents the complete TLA+ description of *SimplePaxos* and *ImplicitPaxos*.

Algorithm 10 *SimplePaxos* protocol in pretty-printed TLA+

```
CONSTANTS $\texttt{value}, \texttt{acceptor}, \texttt{quorum}$
$\texttt{ballot} \triangleq Nat \cup \{-1\}$
VARIABLES $msg1a, msg1b, msg2a, msg2b, maxBal$
$msg1a \in \texttt{ballot} \rightarrow \text{BOOLEAN}$
$msg1b \in (\texttt{acceptor} \times \texttt{ballot}) \rightarrow \text{BOOLEAN}$
$msg2a \in (\texttt{ballot} \times \texttt{value}) \rightarrow \text{BOOLEAN}$
$msg2b \in (\texttt{acceptor} \times \texttt{ballot} \times \texttt{value}) \rightarrow \text{BOOLEAN}$
$maxBal \in \texttt{acceptor} \rightarrow \texttt{ballot}$
ASSUME $\land \forall Q \in \texttt{quorum} : Q \subseteq \texttt{acceptor}$
       $\land \forall Q_1, Q_2 \in \texttt{quorum} : Q_1 \cap Q_2 \neq \{\}$
$chosenAt(b, v) \triangleq \exists Q \in \texttt{quorum} : \forall A \in Q : msg2b(A, b, v)$
$chosen(v) \triangleq \exists B \in \texttt{ballot} : chosenAt(B, v)$
$showsSafeAtSimplePaxos(q, b, v) \triangleq$
  $\land \forall A \in q : msg1b(A, b)$
  $\land \lor \forall A \in \texttt{acceptor} : \forall M_b \in \texttt{ballot} : \forall M_v \in \texttt{value} :$
              $\neg( A \in q \land msg1b(A, b) \land msg2b(A, M_b, M_v))$
     $\lor \exists M_b \in \texttt{ballot} :$
        $\land \exists A \in q : msg1b(A, b) \land msg2b(A, M_b, v)$
        $\land \forall A \in q : \forall M_{b2} \in \texttt{ballot} : \forall M_{v2} \in \texttt{value} :$
              $msg1b(A, b) \land msg2b(A, M_{b2}, M_{v2}) \rightarrow M_{b2} \leq M_b$
$isSafeAtSimplePaxos(b, v) \triangleq \exists Q \in \texttt{quorum} : showsSafeAtSimplePaxos(Q, b, v)$
$Phase1a(b) \triangleq$
  $\land b \neq -1$
  $\land msg1a' = [msg1a$ EXCEPT $![b] = \top]$
  $\land$ UNCHANGED $msg1b, msg2a, msg2b, maxBal$
$Phase1b(a, b) \triangleq$
  $\land b \neq -1 \land msg1a(b) \land b > maxBal(a)$
  $\land maxBal' = [maxBal$ EXCEPT $![a] = b]$
  $\land msg1b' = [msg1b$ EXCEPT $![a, b] = \top]$
  $\land$ UNCHANGED $msg1a, msg2a, msg2b$
```

(continued)

12 $\mathit{Phase2a}(b, v) \ \triangleq$

$\quad \land\ b \neq\ -1\ \land\ \neg(\exists\, V \in \texttt{value} : \mathit{msg2a}(b, V)\,)$

$\quad \land\ \mathit{isSafeAtSimplePaxos}(b, v)$

$\quad \land\ \mathit{msg2a}' = [\mathit{msg2a}\ \text{EXCEPT}\ ![b, v] = \top]$

$\quad \land\ \text{UNCHANGED}\ \mathit{msg1a}, \mathit{msg1b}, \mathit{msg2b}, \mathit{maxBal}$

13 $\mathit{Phase2b}(a, b, v) \ \triangleq$

$\quad \land\ b \neq\ -1\ \land\ \mathit{msg2a}(b, v)\ \land\ b \geq \mathit{maxBal}(a)$

$\quad \land\ \mathit{maxBal}' \ = [\mathit{maxBal}\ \text{EXCEPT}\ ![a] = b]$

$\quad \land\ \mathit{msg2b}' \quad = [\mathit{msg2b}\ \text{EXCEPT}\ ![a, b, v] = \top]$

$\quad \land\ \text{UNCHANGED}\ \mathit{msg1a}, \mathit{msg1b}, \mathit{msg2a}$

14 $\mathit{Init}\ \triangleq\ \forall\, A \in \texttt{acceptor} : B \in \texttt{ballot} :$

$\qquad\qquad \land\ \neg \mathit{msg1a}(B)$

$\qquad\qquad \land\ \neg \mathit{msg1b}(A, B)$

$\qquad\qquad \land\ \forall\, V \in \texttt{value} : \neg \mathit{msg2a}(B, V)\ \land\ \neg \mathit{msg2b}(A, B, V)$

$\qquad\qquad \land\ \mathit{maxBal}(A) = -1$

15 $T\ \triangleq\ \exists\, A \in \texttt{acceptor} : B \in \texttt{ballot} : V \in \texttt{value} :$

$\qquad\qquad \lor\ \mathit{Phase1a}(B) \qquad \lor\ \mathit{Phase1b}(A, B)$

$\qquad\qquad \lor\ \mathit{Phase2a}(B, V)\ \lor\ \mathit{Phase2b}(A, B, V)$

16 $P\ \triangleq\ \forall\, V_1, V_2 \in \texttt{value} : \mathit{chosen}(V_1) \land \mathit{chosen}(V_2) \rightarrow V_1 = V_2$

Algorithm 10 *ImplicitPaxos* protocol in pretty-printed TLA+

```
$\textsf{CONSTANTS}\ \texttt{value}, \texttt{acceptor}, \texttt{quorum}$
$\texttt{ballot} \triangleq Nat \cup \{-1\}$
$\textsf{VARIABLES}\ msg1a, msg1b, msg2a, msg2b, maxBal$
$msg1a \quad \in \texttt{ballot} \rightarrow \textsf{BOOLEAN}$
$msg1b \quad \in (\texttt{acceptor} \times \texttt{ballot} \times \texttt{ballot} \times \texttt{value}) \rightarrow \textsf{BOOLEAN}$
$msg2a \quad \in (\texttt{ballot} \times \texttt{value}) \rightarrow \textsf{BOOLEAN}$
$msg2b \quad \in (\texttt{acceptor} \times \texttt{ballot} \times \texttt{value}) \rightarrow \textsf{BOOLEAN}$
$maxBal \quad \in \texttt{acceptor} \rightarrow \texttt{ballot}$
$\textsf{ASSUME} \quad \land\ \forall\, Q \in \texttt{quorum} : Q \subseteq \texttt{acceptor}$
$\qquad\qquad\quad \land\ \forall\, Q_1, Q_2 \in \texttt{quorum} : Q_1 \cap Q_2 \neq \{\}$
$chosenAt(b, v) \triangleq \exists\, Q \in \texttt{quorum} : \forall\, A \in Q : msg2b(A, b, v)$
$chosen(v) \triangleq \exists\, B \in \texttt{ballot} : chosenAt(B, v)$
$showsSafeAtPaxos(q, b, v) \triangleq$
  $\land\ \forall\, A \in q : \exists\, M_b \in \texttt{ballot} : \exists\, M_v \in \texttt{value} : msg1b(A, b, M_b, M_v)$
  $\land\ \lor\ \forall\, A \in \texttt{acceptor} : \forall\, M_b \in \texttt{ballot} : \forall\, M_v \in \texttt{value} :$
              $\neg(\,A \in q\ \land\ msg1b(A, b, M_b, M_v)\ \land\ (M_b \neq\ -1)\,)$
      $\lor\ \exists\, M_b \in \texttt{ballot} :$
          $\land\ \exists\, A \in q : msg1b(A, b, M_b, v)\ \land\ (M_b \neq\ -1)$
          $\land\ \forall\, A \in q : \forall\, M_{b2} \in \texttt{ballot} : \forall\, M_{v2} \in \texttt{value} :$
              $msg1b(A, b, M_{b2}, M_{v2})\ \land\ (M_{b2} \neq\ -1) \rightarrow M_{b2} \leq M_b$
$isSafeAtPaxos(b, v) \triangleq \exists\, Q \in \texttt{quorum} : showsSafeAtPaxos(Q, b, v)$
$Phase1a(b) \triangleq$
  $\land\ b \neq\ -1$
  $\land\ msg1a' = [msg1a\ \textsf{EXCEPT}\ ![b] = \top]$
  $\land\ \textsf{UNCHANGED}\ msg1b, msg2a, msg2b, maxBal$
$Phase1b(a, b) \triangleq$
  $\land\ b \neq\ -1\ \land\ msg1a(b)\ \land\ b > maxBal(a)$
  $\land\ maxBal' = [maxBal\ \textsf{EXCEPT}\ ![a] = b]$
  $\land\ \exists\, M_b \in \texttt{ballot} : \exists\, M_v \in \texttt{value} :$
      $\land\ \ \lor\ \ \land\ (M_b = -1)$
                 $\land\ \forall\, B \in \texttt{ballot} : \forall\, V \in \texttt{value} : \neg msg2b(a, B, V)$
            $\lor\ \ \land\ (M_b \neq -1)\ \land\ msg2b(a, M_b, M_v)$
                 $\land\ \forall\, B \in \texttt{ballot} : \forall\, V \in \texttt{value} : msb2b(a, B, V) \rightarrow B \leq M_b$
      $\land\ msg1b' = [msg1b\ \textsf{EXCEPT}\ ![a, b, M_b, M_v] = \top]$
  $\land\ \textsf{UNCHANGED}\ msg1a, msg2a, msg2b$
```

(continued)

```
12 $Phase2a(b, v) \triangleq$
     $\land\ b \neq -1 \ \land\ \neg(\exists\, V \in \texttt{value} : msg2a(b, V)\,)$
     $\land\ isSafeAtPaxos(b, v)$
     $\land\ msg2a' = [msg2a\ \textsf{EXCEPT}\ ![b, v] = \top]$
     $\land\ \textsf{UNCHANGED}\ msg1a, msg1b, msg2b, maxBal$

13 $Phase2b(a, b, v) \triangleq$
     $\land\ b \neq -1 \ \land\ msg2a(b, v) \ \land\ b \geq maxBal(a)$
     $\land\ maxBal' \ = [maxBal\ \textsf{EXCEPT}\ ![a] = b]$
     $\land\ msg2b' \ \ = [msg2b\ \textsf{EXCEPT}\ ![a, b, v] = \top]$
     $\land\ \textsf{UNCHANGED}\ msg1a, msg1b, msg2a$

14 $Init \triangleq \forall\, A \in \texttt{acceptor} : B \in \texttt{ballot} :$
                $\land\ \neg msg1a(B)$
                $\land\ \forall\, M_b \in \texttt{ballot} : M_v \in \texttt{value} : \neg msg1b(A, B, M_b, M_v)$
                $\land\ \forall\, V \in \texttt{value} : \neg msg2a(B, V) \ \land\ \neg msg2b(A, B, V)$
                $\land\ maxBal(A) = -1$

15 $T \triangleq \exists\, A \in \texttt{acceptor} : B \in \texttt{ballot} : V \in \texttt{value} :$
                $\lor\ Phase1a(B) \quad\ \lor\ Phase1b(A, B)$
                $\lor\ Phase2a(B, V) \ \lor\ Phase2b(A, B, V)$

16 $P \triangleq \forall\, V_1, V_2 \in \texttt{value} : chosen(V_1) \land chosen(V_2) \rightarrow V_1 = V_2$
```

BIBLIOGRAPHY

[1] ABC: System for Sequential Logic Synthesis and Formal Verification. https://github.com/berkeley-abc/abc.

[2] Apache HTTP server project. https://httpd.apache.org, .

[3] National Vulnerability Database - CVE-2006-3747. https://nvd.nist.gov/vuln/detail/CVE-2006-3747, .

[4] National Vulnerability Database - CVE-2004-0940. https://nvd.nist.gov/vuln/detail/CVE-2004-0940, .

[5] Experiments. https://github.com/aman-goel/tacas20ae.

[6] Btor2Tools. https://github.com/Boolector/btor2tools, .

[7] Btormc bounded model checke. https://github.com/Boolector/boolector/blob/master/src/btormc.c, .

[8] BtorSIM. https://github.com/Boolector/btor2tools/tree/master/src/btorsim, .

[9] Cardano blockchain platform. https://cardano.org.

[10] Coreir symbolic analyzer 2. https://github.com/upscale-project/cosa2.

[11] A configurable C++ generator of pipelined Verilog FFT cores. https://github.com/ZipCPU/dblclockfft.

[12] pySMT: A library for SMT formulae manipulation and solving. https://github.com/aman-goel/pysmt.

[13] The ivy language and verifier. http://microsoft.github.io/ivy, .

[14] A collection of distributed protocol verification problems. https://github.com/aman-goel/ivybench, .

[15] A TLA+ specification of the MultiPaxos algorithm. https://github.com/tlaplus/Examples/tree/master/specifications/MultiPaxos.

[16] mypyvy (github). https://github.com/wilcoxjay/mypyvy.

[17] The nuXmv model checker. `https://nuxmv.fbk.eu`.

[18] PicoRV32 - A Size-Optimized RISC-V CPU. `https://github.com/cliffordwolf/picorv32`.

[19] Pono: a flexible and extensible smt-based model checker. `https://github.com/upscale-project/pono`, .

[20] PonyLink: A single-wire bi-directional chip-to-chip interface for FPGAs. `https://github.com/cliffordwolf/PonyLink`, .

[21] A set of Wishbone Controlled SPI Flash Controllers. `https://github.com/ZipCPU/qspiflash`.

[22] The TLA+ Toolbox. `https://lamport.azurewebsites.net/tla/toolbox.html`.

[23] Toy consensus protocol. `https://github.com/microsoft/ivy/blob/master/examples/ivy/toy_consensus.ivy`.

[24] VexRiscV: A FPGA friendly 32 bit RISC-V CPU implementation. `https://github.com/SpinalHDL/VexRiscv`.

[25] A Video display simulator. `https://github.com/ZipCPU/vgasim`.

[26] Verification Modulo Theories. `http://www.vmt-lib.org`.

[27] Zip CPU: a small, light-weight, RISC CPU. `https://github.com/ZipCPU/zipcpu`.

[28] IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 1–1315, Feb 2018. doi:10.1109/IEEESTD.2018.8299595.

[29] IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 1–1315, Feb 2018. doi:10.1109/IEEESTD.2018.8299595.

[30] Martin Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[31] Parosh Abdulla, Frédéric Haziza, and Lukáš Holík. Parameterized verification through view abstraction. *International Journal on Software Tools for Technology Transfer*, 18(5):495–516, 2016.

[32] Andreas Abel, Marcin Benke, Ana Bove, John Hughes, and Ulf Norell. Verifying haskell programs using constructive type theory. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 62–73, 2005.

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

[34] Zaher S. Andraus, Mark H. Liffiton, and Karem A. Sakallah. Reveal: A Formal Verification Tool for Verilog Designs. In *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'08)*, volume LNCS 5330, pages 343–352, Doha, Qatar, November 2008. Springer.

[35] Krzysztof R. Apt and Dexter Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.

[36] Tamarah Arons, Amir Pnueli, Sitvanit Ruah, Ying Xu, and Lenore Zuck. Parameterized verification with automatically computed inductive assertions. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, pages 221–234, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-44585-2.

[37] Krste Asanović and David A Patterson. Instruction sets should be free: The case for risc-v. 2014.

[38] Domagoj Babić and Alan J Hu. Structural abstraction of software verification conditions. In *International Conference on Computer Aided Verification*, pages 366–378. Springer, 2007.

[39] Ittai Balaban, Yi Fang, Amir Pnueli, and Lenore D Zuck. Iiv: An invisible invariant verifier. In *International Conference on Computer Aided Verification*, pages 408–412. Springer, 2005.

[40] Thomas Ball and Sriram K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 1–3, New York, NY, USA, 2002. ACM. ISBN 1-58113-450-9. doi:10.1145/503272.503274. URL http://doi.acm.org/10.1145/503272.503274.

[41] Thomas Ball, Andreas Podelski, and SriramK. Rajamani. Boolean and Cartesian Abstraction for Model Checking C Programs. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 268–283. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-41865-8. doi:10.1007/3-540-45319-9_19. URL http://dx.doi.org/10.1007/3-540-45319-9_19.

[42] Sharon Barner and Orna Grumberg. Combining symmetry reduction and under-approximation for symbolic model checking. In *International Conference on Computer Aided Verification*, pages 93–106. Springer, 2002.

[43] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. The Coq Proof Assistant Reference Manual : Version 6.1. Research Report RT-0203, INRIA, May 1997. URL https://hal.inria.fr/inria-00069968. Projet COQ.

[44] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[45] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. URL http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf. Snowbird, Utah.

[46] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.

[47] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.

[48] Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification. http://www.eecs.berkeley.edu/~alanmi/abc/, 2017.

[49] Idan Berkovits, Marijana Lazić, Giuliano Losa, Oded Padon, and Sharon Shoham. Verification of threshold-based distributed algorithms by decomposition to decidable logics. In *International Conference on Computer Aided Verification*, pages 245–266. Springer, 2019.

[50] Armin Biere and Mathias Preiner. Hardware model checking competition (HWMCC) 2019. http://fmv.jku.at/hwmcc19.

[51] Armin Biere and Mathias Preiner. Hardware model checking competition 2019. In *2019 Formal Methods in Computer Aided Design (FMCAD)*, 2019.

[52] Armin Biere, Nils Froleyks, and Mathias Preiner. Hardware model checking competition (HWMCC) 2020. http://fmv.jku.at/hwmcc20, .

[53] Armin Biere, Nils Froleyks, and Mathias Preiner. Hardware model checking competition (HWMCC) 2020 results. http://fmv.jku.at/hwmcc20/hwmcc20slides.pdf, .

[54] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, 1999. Springer-Verlag. ISBN 3-540-65703-7. URL http://dl.acm.org/citation.cfm?id=646483.691738.

[55] Armin Biere, Tom van Dijk, and Keijo Heljanko. Hardware model checking competition 2017. In *FMCAD*, pages 9–9, 2017.

[56] Johannes Birgmeier, Aaron R Bradley, and Georg Weissenbacher. Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR). In *International Conference on Computer Aided Verification*, pages 831–848. Springer, 2014.

[57] Nikolaj Bjørner and Arie Gurfinkel. Property directed polyhedral abstraction. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 263–281. Springer, 2015.

[58] Aaron R. Bradley. SAT-Based Model Checking without Unrolling. In *Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation*, VMCAI'11, pages 70–87, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-18274-7. URL http://dl.acm.org/citation.cfm?id=1946284.1946291.

[59] Aaron R. Bradley and Zohar Manna. Checking Safety by Inductive Generalization of Counterexamples to Induction. In *Formal Methods in Computer Aided Design (FMCAD'07)*, pages 173 –180, Nov. 2007. doi:10.1109/FAMCAD.2007.15.

[60] Robert Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification*, pages 24–40. Springer, 2010.

[61] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[62] Denis Bueno and Karem A. Sakallah. euforia: Complete Software Model Checking with Uninterpreted Functions. In *20th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume LNCS 11388, pages 363–385, Cascais, Portugal, January 2019. Springer.

[63] Jerry R. Burch and David L. Dill. Automatic Verification of Pipelined Microprocessor Control. In David L. Dill, editor, *Conference on Computer-Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80. Springer-Verlag, 1994. Stanford, California, June 21–23, 1994.

[64] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.

[65] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic Model checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[66] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, 2006.

[67] Gianpiero Cabodi, Sergio Nocco, and Stefano Quer. The PdTRAV tool. http://fmgroup.polito.it/index.php/download/viewcategory/3-pdtrav-package.

[68] Gianpiero Cabodi, Carmelo Loiacono, Marco Palena, Paolo Pasini, Denis Patti, Stefano Quer, Danilo Vendraminetto, Armin Biere, and Keijo Heljanko. Hardware model checking

competition 2014: An analysis and comparison of solvers and benchmarks. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:135–172, 2016.

[69] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In *CAV*, pages 334–342, 2014.

[70] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In *CAV*, pages 334–342, 2014.

[71] Manuel MT Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The extended utxo model. In *International Conference on Financial Cryptography and Data Security*, pages 525–539. Springer, 2020.

[72] Manuel MT Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann Müller, Michael Peyton Jones, Polina Vinogradova, and Philip Wadler. Native custom tokens in the extended utxo model. In *International Symposium on Leveraging Applications of Formal Methods*, pages 89–111. Springer, 2020.

[73] Saksham Chand, Yanhong A Liu, and Scott D Stoller. Formal verification of multi-paxos for distributed consensus. In *International Symposium on Formal Methods*, pages 119–136. Springer, 2016.

[74] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, page 398–407, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936165. doi:10.1145/1281100.1281103. URL `https://doi.org/10.1145/1281100.1281103`.

[75] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. The tla+ proof system: Building a heterogeneous verification platform. In *International Colloquium on Theoretical Aspects of Computing*, pages 44–44. Springer, 2010.

[76] Hana Chockler, Alexander Ivrii, Arie Matsliah, Shiri Moran, and Ziv Nevo. Incremental formal verification of hardware. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, pages 135–143. FMCAD Inc, 2011.

[77] Alessandro Cimatti and Alberto Griggio. Software model checking via ic3. In *International Conference on Computer Aided Verification*, pages 277–293. Springer, 2012.

[78] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In Nir Piterman and Scott Smolka, editors, *Proceedings of TACAS*, volume 7795 of *LNCS*. Springer, 2013.

[79] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Ic3 modulo theories via implicit predicate abstraction. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 46–61. Springer, 2014.

[80] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Infinite-state invariant checking with ic3 and predicate abstraction. *Formal Methods in System Design*, 49(3):190–218, 2016.

[81] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In E. Emerson and Aravinda Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer Berlin / Heidelberg, 2000. ISBN 978-3-540-67770-3. URL `http://dx.doi.org/10.1007/10722167_15`. 10.1007/10722167_15.

[82] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded Model Checking Using Satisfiability Solving. *Form. Methods Syst. Des.*, 19:7–34, July 2001. ISSN 0925-9856. doi:10.1023/A:1011276507260. URL `http://dl.acm.org/citation.cfm?id=510986.510987`.

[83] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In *Informatics*, pages 176–194. Springer, 2001.

[84] Edmund Clarke, Anubhav Gupta, James Kukula, and Ofer Strichman. SAT Based Abstraction-Refinement Using ILP and Machine Learning Techniques. In *Computer Aided Verification*, pages 265–279. Springer, 2002.

[85] Edmund M. Clarke. The Birth of Model Checking. In *25 Years of Model Checking*, pages 1–26, 2008.

[86] Edmund M Clarke and Orna Grumberg. Avoiding the state explosion problem in temporal logic model checking. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 294–303, 1987.

[87] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach. In *POPL*, pages 117–126, 1983.

[88] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[89] Edmund M Clarke, Orna Grumberg, and Michael C Browne. Reasoning about networks with many identical finite-state processes. In *Proceedings of the fifth annual ACM symposium on Principles of distributed computing*, pages 240–248, 1986.

[90] Edmund M Clarke, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. In *International Conference on Computer Aided Verification*, pages 450–462. Springer, 1993.

[91] Edmund M Clarke, E Allen Emerson, and Joseph Sifakis. Model checking: algorithmic verification and debugging. *Communications of the ACM*, 52(11):74–84, 2009.

[92] Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model checking*. MIT press, 2018.

[93] Michael A Colón and Tomás E Uribe. Generating Finite-State Abstractions of Reactive Systems using Decision Procedures. In *Computer Aided Verification*, pages 293–304. Springer, 1998.

[94] Sylvain Conchon, Amit Goel, Sava Krstić, Alain Mebsout, and Fatiha Zaïdi. Cubicle: A parallel smt-based model checker for parameterized systems. In *International Conference on Computer Aided Verification*, pages 718–724. Springer, 2012.

[95] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. Tla + proofs. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods*, pages 147–154, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-32759-9.

[96] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. doi:10.1145/512950.512973. URL http://doi.acm.org.proxy.lib.umich.edu/10.1145/512950.512973.

[97] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[98] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[99] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[100] Leonardo de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification. In Warren A. Hunt and Fabio Somenzi, editors, *Computer Aided Verification*, pages 14–26, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-45069-6.

[101] Roberto De Prisco, Butler Lampson, and Nancy Lynch. Revisiting the paxos algorithm. *Theoretical Computer Science*, 243(1-2):35–91, 2000.

[102] Coq development team. The coq proof assistant reference manual. http://coq.inria.fr/distrib/current/refman/.

[103] Damien Doligez, Leslie Lamport, and Stephan Merz. A TLA+ specification of the Paxos consensus algorithm and a TLAPS-checked proof of its correctness. https://github.com/tlaplus/tlapm/blob/master/examples/paxos/Paxos.tla.

[104] Michael Dooley and Fabio Somenzi. Proving parameterized systems safe by generalizing clausal proofs of small instances. In *International Conference on Computer Aided Verification*, pages 292–309. Springer, 2016.

[105] Rohit Dureja and Kristin Yvonne Rozier. Fuseic3: An algorithm for checking large design spaces. In *2017 Formal Methods in Computer Aided Design (FMCAD)*, pages 164–171. IEEE, 2017.

[106] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[107] Bruno Dutertre. Yices 2.2. In *International Conference on Computer Aided Verification*, pages 737–744. Springer, 2014.

[108] Niklas Een, Alan Mishchenko, and Robert Brayton. Efficient Implementation of Property Directed Reachability. In *Formal Methods in Computer Aided Design (FMCAD'11)*, pages 125 – 134, Oct. 2011.

[109] E Allen Emerson and Kedar S Namjoshi. Reasoning about rings. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 85–94, 1995.

[110] E Allen Emerson and A Prasad Sistla. Symmetry and model checking. *Formal methods in system design*, 9(1-2):105–131, 1996.

[111] Yotam M. Y. Feldman, Mooly Sagiv, Sharon Shoham, and James R. Wilcox. Learning the boundary of inductive invariants. *CoRR*, abs/2008.09909, 2020. URL https://arxiv.org/abs/2008.09909.

[112] Yotam MY Feldman, Neil Immerman, Mooly Sagiv, and Sharon Shoham. Complexity and information in invariant inference. *Proceedings of the ACM on Programming Languages*, 4 (POPL):1–29, 2019.

[113] Yotam MY Feldman, James R Wilcox, Sharon Shoham, and Mooly Sagiv. Inferring inductive invariants from phase structures. In *International Conference on Computer Aided Verification*, pages 405–425. Springer, 2019.

[114] John B. Fraleigh. *A First Course in Abstract Algebra*. Addison Wesley Longman, Reading, Massachusetts, 6th edition, 2000.

[115] Marco Gario and Andrea Micheli. Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms. In *SMT workshop*, volume 2015, 2015.

[116] Stephen J Garland and Nancy A Lynch. Using i/o automata for developing distributed systems. *Foundations of component-based systems*, 13(285-312):5–2, 2000.

[117] Steven M German and A Prasad Sistla. Reasoning about systems with many processes. *Journal of the ACM (JACM)*, 39(3):675–735, 1992.

[118] Elaheh Ghassabani, Andrew Gacek, and Michael W Whalen. Efficient generation of inductive validity cores for safety properties. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 314–325, 2016.

[119] Patrice Godefroid. Exploiting symmetry when model-checking software. In *Formal Methods for Protocol Engineering and Distributed Systems*, pages 257–275. Springer, 1999.

[120] A. Goel and K. Sakallah. Empirical evaluation of ic3-based model checking techniques on verilog rtl designs. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 618–621, March 2019. doi:10.23919/DATE.2019.8715289.

[121] Aman Goel and Karem Sakallah. AVR: Abstractly Verifying Reachability. http://www.github.com/aman-goel/avr, .

[122] Aman Goel and Karem Sakallah. Benchmarks for Word-level Model Checking. https://github.com/aman-goel/avr/tree/master/tests, .

[123] Aman Goel and Karem Sakallah. AVR experiments. https://github.com/aman-goel/nfm2019exp, .

[124] Aman Goel and Karem Sakallah. Model checking of verilog rtl using ic3 with syntax-guided abstraction. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods*, pages 166–185, Cham, 2019. Springer International Publishing. ISBN 978-3-030-20652-9.

[125] Aman Goel and Karem Sakallah. Avr: Abstractly verifying reachability. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 413–422. Springer, 2020.

[126] Aman Goel and Karem Sakallah. On symmetry and quantification: A new approach to verify distributed protocols. In Aaron Dutle, Mariano M. Moscato, Laura Titolo, César A. Muñoz, and Ivan Perez, editors, *NASA Formal Methods*, pages 131–150, Cham, 2021. Springer International Publishing. ISBN 978-3-030-76384-8. doi:10.1007/978-3-030-76384-8_9. URL https://doi.org/10.1007/978-3-030-76384-8_9.

[127] Aman Goel and Karem A. Sakallah. IC3PO: IC3 for Proving Protocol Properties. https://github.com/aman-goel/ic3po, .

[128] Aman Goel and Karem A. Sakallah. On symmetry and quantification: A new approach to verify distributed protocols. *CoRR*, abs/2103.14831, 2021. URL https://arxiv.org/abs/2103.14831.

[129] Aman Goel and Karem A. Sakallah. Towards an Automatic Proof of Lamport's Paxos. In Ruzica Piskac and Michael W Whalen, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, pages 112–122, New Haven, Connecticut, October 2021. doi:10.34727/2021/isbn.978-3-85448-046-4_20. URL https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_20.

[130] Aman Goel and Karem A. Sakallah. Towards an automatic proof of lamport's paxos. *CoRR*, abs/2108.08796, 2021. URL https://arxiv.org/abs/2108.08796.

[131] Eugene Goldberg, Matthias Güdemann, Daniel Kroening, and Rajdeep Mukherjee. Efficient verification of multi-property designs (the benefit of wrong assumptions). In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 43–48. IEEE, 2018.

[132] Susanne Graf and Hassen Saidi. Construction of Abstract State Graphs with PVS. In Orna Grumberg, editor, *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer Berlin Heidelberg, 1997. ISBN 978-3-540-63166-8. doi:10.1007/3-540-63166-6_10. URL http://dx.doi.org/10.1007/3-540-63166-6_10.

[133] Tomás Grimm, Djones Lettnin, and Michael Hübner. A survey on formal verification techniques for safety-critical systems-on-chip. *Electronics*, 7(6):81, 2018.

[134] Aarti Gupta, Zijiang Yang, Pranav Ashar, and Anubhav Gupta. Sat-based image computation with application in reachability analysis. In *International Conference on Formal Methods in Computer-Aided Design*, pages 391–408. Springer, 2000.

[135] Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. Quantifiers on demand. In *International Symposium on Automated Technology for Verification and Analysis*, pages 248–266. Springer, 2018.

[136] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. Finding invariants of distributed systems: It's a small (enough) world after all. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 115–131. USENIX Association, April 2021. ISBN 978-1-939133-21-2. URL https://www.usenix.org/conference/nsdi21/presentation/hance.

[137] Zyad Hassan, Aaron R Bradley, and Fabio Somenzi. Better generalization in IC3. In *FMCAD*, pages 157–164, 2013.

[138] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17. ACM, 2015.

[139] Thomas A Henzinger, Shaz Qadeer, and Sriram K Rajamani. You assume, we guarantee: Methodology and case studies. In *International Conference on Computer Aided Verification*, pages 440–451. Springer, 1998.

[140] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software Verification with BLAST. In *Model Checking Software*, pages 235–239. Springer, 2003.

[141] Yen-Sheng Ho, Pankaj Chauhan, Pritam Roy, Alan Mishchenko, and Robert Brayton. Efficient uninterpreted function abstraction and refinement for word-level model checking. In *FMCAD*, pages 65–72, 2016.

[142] Yen-Sheng Ho, Alan Mishchenko, and Robert Brayton. Property directed reachability with word-level abstraction. In *FMCAD*, pages 132–139, 2017.

[143] Yen-Sheng Ho, Alan Mishchenko, Robert Brayton, and Niklas Eén. Enhancing PDR / IC3 with localization abstraction. 2017.

[144] Kryštof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 157–171. Springer, 2012.

[145] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited. *CoRR*, abs/1608.06696, 2016. URL `http://arxiv.org/abs/1608.06696`.

[146] C Norris Ip and David L Dill. Better verification through symmetry. In *Computer Hardware Description Languages and their Applications*, pages 97–111. Elsevier, 1993.

[147] Ahmed Irfan, Alessandro Cimatti, Alberto Griggio, Marco Roveri, and Roberto Sebastiani. Verilog2SMV: A Tool for Word-Level Verification. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pages 1156–1159. IEEE, 2016.

[148] Michael Isard. Autopilot: Automatic data center management. *SIGOPS Oper. Syst. Rev.*, 41(2):60–67, April 2007. ISSN 0163-5980. doi:10.1145/1243418.1243426. URL `https://doi.org/10.1145/1243418.1243426`.

[149] Himanshu Jain, Daniel Kroening, Natasha Sharygina, and Edmund Clarke. Vcegar: Verilog counterexample guided abstraction refinement. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 583–586. Springer, 2007.

[150] Christopher Jefferson, Angela Miguel, Ian Miguel, and S Armagan Tarim. Modelling and solving english peg solitaire. *Computers & Operations Research*, 33(10):2935–2959, 2006.

[151] Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Property-directed inference of universal invariants or proving their absence. *J. ACM*, 64(1), March 2017. ISSN 0004-5411. doi:10.1145/3022187. URL `https://doi.org/10.1145/3022187`.

[152] Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Property-directed inference of universal invariants or proving their absence. *Journal of the ACM (JACM)*, 64(1):1–33, 2017. doi:https://doi.org/10.1145/3022187. URL `https://bitbucket.org/tausigplan/updr-distrib/src/master/`.

[153] Jason R. Koenig, Oded Padon, Neil Immerman, and Alex Aiken. First-order quantified separators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 703–717, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi:10.1145/3385412.3386018. URL `https://github.com/wilcoxjay/mypyvy/tree/pldi20-artifact`.

[154] Jason R. Koenig, Oded Padon, Neil Immerman, and Alex Aiken. First-order quantified separators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language*

Design and Implementation, PLDI 2020, page 703–717, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi:10.1145/3385412.3386018. URL `https://doi.org/10.1145/3385412.3386018`.

[155] Kenneth Kotovsky, John R Hayes, and Herbert A Simon. Why are some problems hard? evidence from tower of hanoi. *Cognitive psychology*, 17(2):248–294, 1985.

[156] Bernhard Kragl, Shaz Qadeer, and Thomas A Henzinger. Refinement for structured concurrent programs. In *International Conference on Computer Aided Verification*, pages 275–298. Springer, 2020.

[157] Hari Govind Vediramana Krishnan, Yakir Vizel, Vijay Ganesh, and Arie Gurfinkel. Interpolating strong induction. In *International Conference on Computer Aided Verification*, pages 367–385. Springer, 2019.

[158] Kelvin Ku, Thomas E Hart, Marsha Chechik, and David Lie. A buffer overflow benchmark for software model checkers. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 389–392. ACM, 2007.

[159] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, Princeton, NJ, USA, 1994. ISBN 0-691-03436-2.

[160] Robert P Kurshan and Ken McMillan. A structural induction theorem for processes. In *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 239–247, 1989.

[161] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE transactions on software engineering*, (2):125–143, 1977.

[162] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

[163] Leslie Lamport. How to write a proof. *The American mathematical monthly*, 102(7):600–608, 1995.

[164] Leslie Lamport. Refinement in state-based formalisms. *Digital Equipment Corportation*, 1996.

[165] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[166] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998. ISSN 0734-2071. doi:10.1145/279227.279229. URL `https://doi.org/10.1145/279227.279229`.

[167] Leslie Lamport. Paxos made simple. pages 51–58, December 2001. URL `https://www.microsoft.com/en-us/research/publication/paxos-made-simple/`.

[168] Leslie Lamport. *Specifying Systems*, volume 388. Addison-Wesley Boston, 2002.

[169] Leslie Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, March 2005. URL https://www.microsoft.com/en-us/research/publication/generalized-consensus-and-paxos/.

[170] Leslie Lamport. Byzantizing paxos by refinement. In *International Symposium on Distributed Computing*, pages 211–224. Springer, 2011.

[171] Leslie Lamport. A TLA+ specification of the Paxos Consensus algorithm from Leslie Lamport's lectures titled: The Paxos Algorithm - or How to Win a Turing Award. https://github.com/tlaplus/Examples/blob/master/specifications/PaxosHowToWinATuringAward/Paxos.tla, 2019.

[172] Leslie Lamport. The Paxos Algorithm - or How to Win a Turing Award. https://lamport.azurewebsites.net/tla/paxos-algorithm.html?back-link=more-stuff.html, 2019.

[173] Leslie Lamport. A TLA+ specification of the Voting algorithm from Leslie Lamport's lectures titled: The Paxos Algorithm - or How to Win a Turing Award. https://github.com/tlaplus/Examples/blob/master/specifications/PaxosHowToWinATuringAward/Voting.tla, 2019.

[174] Leslie Lamport and Stephan Merz. A TLA+ specification of the Voting algorithm and a TLAPS-checked proof of its correctness. https://github.com/tlaplus/tlapm/blob/master/examples/ByzPaxos/VoteProof.tla.

[175] Tim Lange, Martin R Neuhauber, and Thomas Noll. Ic3 software model checking on control flow automata. In *2015 Formal Methods in Computer-Aided Design (FMCAD)*, pages 97–104. IEEE, 2015.

[176] Suho Lee. Unbounded scalable hardware verification. 2016.

[177] Suho Lee and Karem A Sakallah. Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction. In *CAV*, pages 849–865, 2014.

[178] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-17510-4, 978-3-642-17510-7. URL http://dl.acm.org/citation.cfm?id=1939141.1939161.

[179] Harry R Lewis. Complexity results for classes of quantificational formulas. *Journal of Computer and System Sciences*, 21(3):317–353, 1980.

[180] Yongjian Li, Jun Pang, Yi Lv, Dongrui Fan, Shen Cao, and Kaiqiang Duan. Paraverifier: An automatic framework for proving parameterized cache coherence protocols. In *International Symposium on Automated Technology for Verification and Analysis*, pages 207–213. Springer, 2015.

[181] Mark H Liffiton and Karem A Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008.

[182] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, December 1975. ISSN 0001-0782. doi:10.1145/361227.361234. URL http://doi.acm.org/10.1145/361227.361234.

[183] Giuliano Losa. Paxos consensus protocol in Ivy. https://github.com/nano-o/ivy-proofs/blob/master/paxos/paxos.ivy.

[184] Gavin Lowe. An attack on the needham- schroeder public- key authentication protocol. *Information processing letters*, 56(3), 1995.

[185] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A Sakallah. I4: Incremental inference of inductive invariants for verification of distributed protocols. https://github.com/GLaDOS-Michigan/I4.

[186] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. I4: Incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th Symposium on Operating Systems Principles*. ACM, 2019.

[187] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A Sakallah. Towards automatic inference of inductive invariants. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. ACM, 2019.

[188] Makai Mann, Ahmed Irfan, Alberto Griggio, Oded Padon, and Clark Barrett. Counterexample-guided prophecy for model checking modulo the theory of arrays. *arXiv preprint arXiv:2101.06825*, 2021.

[189] Makai Mann, Ahmed Irfan, Florian Lonsing, Yahan Yang, Hongce Zhang, Kristopher Brown, Aarti Gupta, and Clark Barrett. Pono: a flexible and extensible smt-based model checker. In *International Conference on Computer Aided Verification*, pages 461–474. Springer, 2021.

[190] Jõao Marques-Silva and Karem A. Sakallah. GRASP-A New Search Algorithm for Satisfiability. In Andreas Kuehlmann, editor, *The Best of ICCAD: 20 Years of Excellence in Computer-Aided Design*, pages 73–89. Kluwer Academic Publishers, 2003.

[191] João P. Marques-Silva and Karem A. Sakallah. GRASP-A New Search Algorithm for Satisfiability. In *Digest of IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 220–227, San Jose, California, November 1996.

[192] João P. Marques-Silva and Karem A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

[193] Cristian Mattarei, Makai Mann, Clark Barrett, Ross G Daly, Dillon Huff, and Pat Hanrahan. Cosa: Integrated verification for agile hardware design. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–5. IEEE, 2018.

[194] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993. ISBN 0792393805.

[195] Kenneth L McMillan. Applications of craig interpolants in model checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–12. Springer, 2005.

[196] Kenneth L McMillan. Interpolants and symbolic model checking. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 89–90. Springer, 2007.

[197] Kenneth L McMillan and Oded Padon. Deductive verification in decidable fragments with ivy. In *International Static Analysis Symposium*, pages 43–55. Springer, 2018.

[198] Stephan Merz. Formal specification and verification. In *Concurrency: the Works of Leslie Lamport*, pages 103–129. 2019.

[199] Alan Mishchenko, Michael Case, Robert Brayton, and Stephen Jang. Scalable and scalably-verifiable sequential synthesis. In *2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 234–241. IEEE, 2008.

[200] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC*, pages 530–535, 2001.

[201] Rajdeep Mukherjee, Michael Tautschnig, and Daniel Kroening. v2c–a verilog to c translator. In *TACAS*, pages 580–586, 2016.

[202] Kedar S Namjoshi. Symmetry and completeness in the analysis of parameterized systems. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 299–313. Springer, 2007.

[203] Roger M Needham and Michael D Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[204] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:53–58, 2014 (published 2015).

[205] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2, btormc and boolector 3.0. In *International Conference on Computer Aided Verification*, pages 587–595. Springer, 2018.

[206] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[207] C. Norris IP and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1):41–75, Aug 1996. ISSN 1572-8102. doi:10.1007/BF00625968. URL `https://doi.org/10.1007/BF00625968`.

[208] Yoonna Oh, Maher N Mneimneh, Zaher S Andraus, Karem A Sakallah, and Igor L Markov. Amuse: a minimally-unsatisfiable subformula extractor. In *Proceedings of the 41st annual Design Automation Conference*, pages 518–523. ACM, 2004.

[209] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.

[210] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 614–630, 2016.

[211] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made epr: decidable reasoning about distributed protocols. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):108:1–108:31, 2017.

[212] Radek Pelánek. Beem: benchmarks for explicit model checkers. In *International SPIN Workshop on Model Checking of Software*, pages 263–267. Springer, 2007.

[213] Sebastiano Peluso, Alexandru Turcu, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Making fast consensus generally faster. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 156–167. IEEE, 2016.

[214] Ruzica Piskac, Leonardo de Moura, and Nikolaj Bjørner. Deciding effectively propositional logic using dpll and substitution sets. *Journal of Automated Reasoning*, 44(4):401–424, 2010.

[215] Amir Pnueli, Sitvanit Ruah, and Lenore Zuck. Automatic deductive verification with invisible invariants. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 82–97. Springer, 2001.

[216] Fong Pong and Michel Dubois. A new approach for the verification of cache coherence protocols. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):773–787, 1995.

[217] Silvio Ranise and Silvio Ghilardi. Backward reachability of array-based systems by smt solving: Termination and invariant synthesis. *Logical Methods in Computer Science*, 6, 2010.

[218] Gian-Carlo Rota. The number of partitions of a set. *The American Mathematical Monthly*, 71(5):498–504, 1964.

[219] Tobias Seufert and Christoph Scholl. Combining pdr and reverse pdr for hardware model checking. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 49–54. IEEE, 2018.

[220] Tobias Seufert and Christoph Scholl. fbpdr: In-depth combination of forward and backward analysis in property directed reachability. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 456–461. IEEE, 2019.

[221] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In *International conference on formal methods in computer-aided design*, pages 127–144. Springer, 2000.

[222] A Prasad Sistla, Viktor Gyuris, and E Allen Emerson. Smc: a symmetry-based model checker for verification of safety and liveness properties. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(2):133–166, 2000.

[223] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1):17–32, 2003.

[224] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. Dependent types and multi-monadic effects in f. In *ACM SIGPLAN Notices*, volume 51, pages 256–270. ACM, 2016.

[225] Paul Tafertshofer and Andreas Ganz. Sat based atpg using fast justification and propagation in the implication graph. In *Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pages 139–146. IEEE Press, 1999.

[226] Marcelo Taube, Giuliano Losa, Kenneth L McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R Wilcox, and Doug Woos. Modularity for decidability of deductive verification with applications to distributed systems. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 662–677, 2018.

[227] Donald Thomas and Philip Moorby. *The Verilog® hardware description language*. Springer Science & Business Media, 2008.

[228] Yakir Vizel and Arie Gurfinkel. Interpolating property directed reachability. In *CAV*, pages 260–276, 2014.

[229] Yakir Vizel, Orna Grumberg, and Sharon Shoham. Lazy abstraction and sat-based reachability in hardware model checking. In *FMCAD*, pages 173–181, 2012.

[230] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. The risc-v instruction set manual, volume i: User-level isa. *CS Division, EECE Department, University of California, Berkeley*, 2014.

[231] Tobias Welp and Andreas Kuehlmann. Qf bv model checking with property directed reachability. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 791–796. EDA Consortium, 2013.

[232] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 357–368, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi:10.1145/2737924.2737958. URL http://doi.acm.org/10.1145/2737924.2737958.

[233] Clifford Wolf. Yosys open synthesis suite. http://www.clifford.at/yosys/.

[234] Pierre Wolper and Vinciane Lovinfosse. Verifying properties of large sets of processes with network invariants. In *International Conference on Computer Aided Verification*, pages 68–80. Springer, 1989.

[235] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. Distai: Data-driven automated invariant learning for distributed protocols. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 405–421, 2021.

[236] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999.

[237] Lenore Zuck and Amir Pnueli. Model checking and abstraction to the aid of parameterized systems (a survey). *Computer Languages, Systems & Structures*, 30(3-4):139–169, 2004.