INVARIANT INFERENCE VIA QUANTIFIED SEPARATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Jason Richard Koenig

December 2021

Preface

Quantified first-order formulas, often with quantifier alternations, are increasingly used in the verification of complex systems. While automated theorem provers for first-order logic are becoming more robust, until recently invariant inference tools that handle quantifiers were restricted to purely universal formulas. We define and analyze first-order quantified separators and their application to inferring quantified invariants with alternations. A *separator* for a given set of positively and negatively labeled structures is a formula that is true on positive structures and false on negative structures. We investigate the problem of finding a separator from the class of formulas in prenex normal form with a bounded number of quantifiers and show this problem is NP-complete by reduction to and from SAT.

Based on this computational primitive, we present a new PDR/IC3-based algorithm for finding inductive invariants with quantifier alternations. We tackle scalability issues that arise due to the large search space of quantified invariants by combining a breadth-first search strategy and a new syntactic form for quantifier-free bodies. The breadth-first strategy prevents inductive generalization from getting stuck in regions of the search space that are expensive to search and focuses instead on lemmas that are easy to discover. The new syntactic form is well-suited to lemmas with quantifier alternations by allowing both limited conjunction and disjunction in the quantifier-free body, while carefully controlling the size of the search space. We evaluate both separation itself and our combined invariant inference algorithm on a benchmark of primarily distributed protocols, including complex Paxos variants. We demonstrate separation can be solved in practice, and that our inference algorithm can solve more of the most complicated examples than other state-of-the-art techniques.

Acknowledgments

I would like to first my deepest thanks to my advisor, Alex, for his support and mentorship. I have learned more collaborating with him than I could have ever imagined as a young PhD student. His experience and guidance has help me grow as a researcher, academic, and writer. I am immensely grateful for all he has done for me throughout this process.

I would like to thank Oded for his inspiration and guidance. His knowledge, expertise, and late deadline-night writing sessions are all appreciated.

I additionally thank my co-authors Sharon and Neil for their collaboration and feedback.

I thank my thesis committee, Alex, Caroline, Clark, Fredrik, and Thomas, for their feedback and patience. I would like to extend a special thanks to Caroline and Clark for taking the time to read and give feedback on my dissertation.

Finally, I would like to thank my parents for instilling in me a lifelong love of learning.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Formal verification of software is the task of proving the correctness of a system with respect to some specification. Most verification tasks require *inductive invariants* for these proofs, but generating these invariants manually is difficult and time consuming. In addition, for complex unbounded systems the required invariants often involve quantifiers, including quantifier alternations. For example, an invariant for a distributed system may need to quantify over an unbounded number of nodes, messages, etc. Furthermore, the invariant may need to nest quantifiers in alternation (between $\forall$ and $\exists$) to capture the system's correctness. For example, one crucial invariant of the Paxos consensus protocol is "every decision must come from a quorum of votes," i.e. $\forall \mathit{decision}. \exists \mathit{quorum}. \forall \mathit{node}.\ \mathit{node} \in \mathit{quorum} \Rightarrow \mathit{node\ voted\ for\ decision}.$ Invariant inference, the problem of automatically generating an invaraint for a given system, is a long-standing problem in formal verification. By avoiding the need to write a invariant in addition to the system, inference reduces the effort required to produce verified systems. We present a solution to the invariant inference problem for invariants with quantifier alternations.

Program invariants are often used as *separators* of states, evaluating to *true* on the good (reachable) states and *false* on the bad (error) states of a system. A common approach in invariant inference algorithms for quantifier-free invariants is to learn a formula that separates good and bad states (e.g., [40]); with enough of the right sort of examples, the discovered separator will hopefully be an invariant.

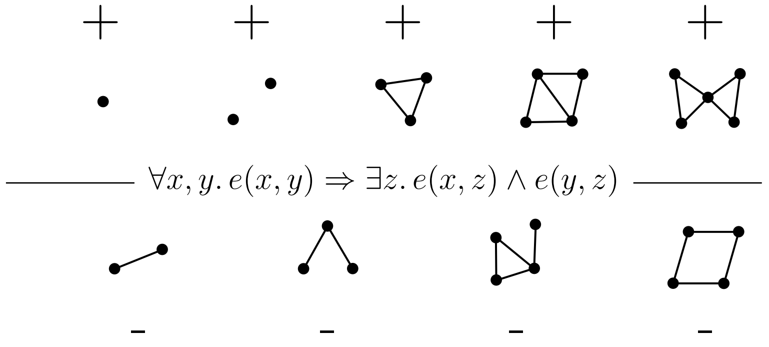
+
+
+
+
+
$\forall x, y.\, e(x,y) \Rightarrow \exists z.\, e(x,z) \wedge e(y,z)$
-
-
-
-

Figure 1.1: A set of labeled structures represented by graphs, and a quantified separator between them. The separator may be interpreted as "every edge is part of a triangle." The structures are defined over a signature with a single sort for vertices and a single symmetric relation $e(\cdot,\cdot)$ for edges.

We introduce the problem of *first-order quantified separation*, which we believe is a key step towards improved automation for verification problems requiring quantified invariants. We give algorithms and complexity results for inferring separators that are quantified first-order formulas. We show how to adapt PDR/IC3 [4] to use separation. Our technique was the first to infer invariants with alternations [28], and is the only system able to infer invariants for five challenging distributed protocols requiring alternations (Section 8.3.2).

An example of a separation query (where structures are represented as undirected graphs) and a possible solution is given in Figure 1.1. The query is the set of labeled structures, and the problem is to generate a formula that is satisfied by the positive structures and is not satisfied by the negative structures. In this query, the structures labeled positive all have edges that form triangles, if they exist. In the negative structures, there is at least one edge that is not part of any triangle. A property that separates these structures is thus "every edge is part of a triangle." To create a concrete formula from this property, we make use of a relation $e(x, y)$ representing when vertices x, y are connected by an edge, and let quantification range over vertices. We can then quantify over edges using a formula of the form $\forall x, y.e(x, y) \Rightarrow (\ldots)$. To express that x, y is part of a triangle, we say "there is a vertex z connected to both x and y," or formally $\exists z.e(x, z) \wedge (y, z)$. Putting this together, we say that

$\forall x, y. e(x, y) \Rightarrow \exists z. e(x, z) \wedge e(y, z)$ is a separator of these structures, and thus a solution to the separation query.

We consider separators expressed in uninterpreted first-order logic. The separability problem has some connections to graph isomorphism [24], Boolean function learning [5], and interpolation [34]. If arbitrary first-order formulas are permitted, then it is trivial to separate any two finite sets of distinct structures. However, generating such separators ends up *overfitting* to the specific structures and fail to generalize to new positive and negative structures. A separation algorithm prone to overfitting will require more labeled structures to generate a specific desired formula, which can significantly increase the cost of invariant inference. We propose formulas with at most k quantifiers in prenex normal form (k-prenex formulas) as an interesting class of separators, and study its theoretical overfitting potential. In particular, the division into a prefix quantifier structure and a quantifier-free body (the *matrix*) is critical for our separation algorithm, which leverages the power of existing SAT solvers.

We show that some other separability problems, such as quantifier-depth k formulas, are in P. We show that for fixed $k \geq 2$, the k-prenex separability problem is NP-complete. One direction of the proof reduces a given SAT problem to a carefully constructed set of highly symmetric structures that force the separator to encode an assignment to the SAT problem in the matrix (Boolean) part of the formula. By using the two player game semantics of first-order logic, we show that the separability of this set of structures is insensitive to permutations of the quantifiers, and depends only on whether the formula has the right *number* of $\forall$ and $\exists$ quantifiers.

The other direction of the NP-completeness proof is given by our algorithm, which reduces separation to SAT. We develop the connection to SAT by showing that the quantifier structure can be eagerly computed up front, leaving only a SAT problem that corresponds to a search for the Boolean part of the formula. While prenex form improves the generalization of the quantifier part of the formula, our algorithm also uses a new form for the Boolean structure, k-term pDNF. This class of Boolean formulas is inspired by the implication structure of human-written invariants and allows for both limited conjunction and disjunction while keeping the search space manageable compared to traditional conjunctive or disjunctive normal forms (CNF or DNF). In particular, many invariants have the general structure “for all objects satisfying some constraints, there exists an object satisfying some other particular properties,” which is the same structure as the formula in Figure 1.1. Many of the lemmas that arise in our evaluation would require many clauses in CNF, but are only 2-term pDNF. We extend separation to search for lemmas of this form, leading to a reduced search space compared to CNF or DNF, resulting in both faster inductive generalization and less overfitting.

Many recent successful invariant inference techniques, including ours, are based on PDR/IC3 [4, 9]. PDR/IC3 progresses by building a collection of *lemmas*, organized into *frames* labeled by number of steps from the initial states, until eventually some of these lemmas form an inductive invariant. New lemmas are generated by *inductive generalization*, where a given (often backward reachable) state is generalized to a formula that excludes it and is inductive *relative* to a previous frame. Inductive generalization therefore plays a key role in PDR/IC3 implementations. Specifically, extending PDR/IC3 to a new domain of lemmas requires a suitable inductive generalization procedure.

While separation can be used naively as a black-box to implement inductive generalization using a refinement loop with an SMT solver [28], this does not scale to challenging protocols such as Paxos and its variants. These examples require invariants with many lemmas (up to 44) and many quantifiers (up to 6 or 7) in each lemma, and the search space for quantified separators explodes as the number of sorts and symbols in the vocabulary and number of quantifiers increases.

When targeting complex invariants, there are two main challenges for inductive generalization: (i) the run time of each individual query; and (ii) overfitting, i.e., learning a lemma that eliminates the given state but does not advance the search for an inductive invariant. We tackle both problems via two strategies: the first integrates inductive generalization with separation in a breadth-first way, and the second leverages our new k-term pDNF form for the quantifier-free Boolean structure of the separators.

Integrating quantified separation with inductive generalization enables us to effectively use a breadth-first rather than a depth-first search strategy for the quantifiers of potential separators: we search in multiple parts of the search space simultaneously rather than exhaustively exploring one region before moving to the next (Section 7.2). Beyond enabling parallelism, and thus faster wall-clock times, this restructuring can change which solution is found by allowing easy-to-search regions to find a solution first. We find that these easier-to-find formulas generalize better (i.e., avoid overfitting), and also result in faster subsequent queries.

We evaluate our technique on a benchmark suite that includes challenging distributed protocols. Inferring invariants with quantifier alternations has recently drawn significant attention, with recent works ([16, 19, 17]) presenting various techniques for this problem. Our experiments show that our separation-based approach significantly advances the state of the art, and scales to three Paxos variants that are unsolved by other techniques. We also present an ablation study that investigates the individual effect of key features of our technique.

Finally, we summarize our findings and discuss the implications of our results and the possibilities for future work. In particular, we identify important steps to make automated invariant inference practical for real systems, such as relaxing the logic restrictions that enable efficient SMT solving, and extending separation to interpreted domains such as arithmetic.

1.1 Overview

This dissertation is structured into three main parts: (1) introducing quantified separation, (2) developing an algorithm to solve separation and presenting a complexity proof of NP-completeness, and (3) presenting a variant of PDR/IC3 that uses separation to infer invariants for complex distributed protocols. Throughout, we will transition from viewing separation as a computational problem on its own, to viewing it as a tool for invariant inference.

We begin in Chapter 2 by presenting the preliminary definitions required to understand our contributions, including a description of the logic we use and our representation of transition systems. This is important because it determines the kinds of systems we can model and thus infer the invariants for. We also present a review of related work. In Chapter 3, we then define the problem of separation itself, and select depth-limited prenex formulas as the class of separators.

In Chapter 4, we present our algorithm for separation, which we introduce gradually. Our algorithm reduces separation to SAT, which forms one half of our complexity proof for the corresponding decision problem. We also present extensions to the algorithm, including using k-pDNF matrices for better generalization and a strategy to limit formulas to the EPR logic fragment. In Chapter 5, we then show separation is NP-hard. Together with the reduction to SAT, this shows our decision problem is NP-complete, and justifies our use of the complexity of a SAT solver.

We consider the problem of invariant inference in Chapter 6. We review the problem of invariant inference, and present two existing inference algorithms, ICE learning and PDR/IC3. In Chapter 7, we present our variant of PDR/IC3 that uses separation to implement inductive generalization, one of the essential components of PDR/IC3. In Chapter 8, we evaluate both separation on its own and our PDR/IC3 variant on a benchmark of difficult invariant inference problems. Finally, Chapter 9 discusses possibilities for future work and summarizes our contributions.

1.2 Contributions

This work makes the following contributions:

1. The introduction of quantified separation as a new computational problem (Section 3.1), and an exploration of the potential classes of formulas used for separation leading to the definition of k-SEP using bounded prenex formulas (Section 3.2.3)

2. An algorithm to solve k-SEP by translation to SAT (Chapter 4)

3. A proof that k-SEP is NP-complete (Chapter 5)

4. A syntactic form of lemmas (k-pDNF, Section 4.5) that is well-suited for invariants with quantifier alternations

5. A variant of PDR/IC3 (Chapter 7) built on quantified separation that can infer the invariants of many complex distributed protocols, some of which are unsolved by other state-of-the-art tools

6. A comprehensive evaluation (Chapter 8) on a large benchmark suite including complex Paxos variants, comparisons with a variety of state-of-the-art tools [19, 16, 17, 27], and an ablation study exploring the effects of key features of our technique

Chapter 2

Background

We begin by presenting essential definitions and concepts from formal logic. We will then explore how to model distributed protocols with the transition systems subject to our restrictions. Finally, we will present a summary of related work, both prior and concurrent.

2.1 Preliminaries

2.1.1 First-Order Logic

We use first-order, many-sorted logic with equality. Formulas in this logic are defined relative to some signature naming the uninterpreted sorts, and the constant, relation, and function symbols and their sorts. We consider only finite signatures, i.e. ones with a finite number of sorts and symbols. *Terms* are constants, variables, or function symbols applied to other terms. Examples include x, $f(y)$, and $g(x, f(z))$. *Atoms* are a relation symbol or equality applied to terms of appropriate sorts, and *literals* are atoms or their negation. Examples of atoms include $p(x)$, $x = y$, and $r(x, f(y))$. Finally, *formulas* are the closure of literals under conjunction, disjunction, negation and quantification. An example formula over a signature of two sorts s_1 and s_2 is $\forall x : s_1.\ \exists y : s_2.\ (p(x) \land \neg r(x, y)) \lor x = f(y)$.

A standard result in logic is that any formula is equivalent to one in *prenex normal form*, in which all quantifiers are lifted to the front of the formula:[1]

$$(\forall x.\, P) \wedge Q \equiv \forall x.\, (P \wedge Q) \tag{2.1}$$
$$(\forall x.\, P) \vee Q \equiv \forall x.\, (P \vee Q) \tag{2.2}$$
$$(\exists x.\, P) \wedge Q \equiv \exists x.\, (P \wedge Q) \tag{2.3}$$
$$(\exists x.\, P) \vee Q \equiv \exists x.\, (P \vee Q) \tag{2.4}$$
$$\neg \forall x.\, P \equiv \exists x.\, \neg P \tag{2.5}$$
$$\neg \exists x.\, P \equiv \forall x.\, \neg P \tag{2.6}$$

For such *prenex* formulas, the quantifier part is called the *prefix* and the remaining Boolean structure is called the *matrix* (usually denoted φ).

A *structure* M over some signature S, is a set of sorted elements, along with a well-sorted interpretation for each constant, relation and function symbol of S. We only consider *finite structures*, i.e. ones in which the set of elements is finite. We assume all sorts have at least one element. An *assignment* is a mapping, σ, from variables to elements of M. A structure M may be augmented with σ to form $M \cup \sigma$ by adding the variables as new constants interpreted according to σ. We say that a structure M *satisfies* a formula p, written $M \models p$, if p is true when its symbols are interpreted according to M. In this case, M is known as a *model* of p.

2.1.2 Game Semantics of Logic

The standard semantics for first-order logic may be defined via a two player game, with one player as $\forall$ and one as $\exists$ [21]. We use a semantics simplified to answer $M \models p$ for prenex formulas p with matrix φ: the two players take turns picking appropriately sorted elements according to their order in the prefix, with the game ending in some assignment σ when the prefix is exhausted. The $\exists$ player wins if and

[1]We assume alpha-renaming to avoid capturing free occurrences of the quantifed variable when lifting a quantifer, and that all sorts are non-empty.

only if $M \cup \sigma \models \varphi$. The semantics becomes: $M \models p$ if and only if $\exists$ has a winning strategy; otherwise $\forall$ has a winning strategy and we say $M \not\models p$. The best-known algorithms for deciding $M \models p$ are exponential in the number of quantifiers of p, as in the worst case they explore all the exponentially many assignments of quantified variables.

Consider the structure M with three elements e_1, e_2, e_3 and a relation p defined by $\neg p(e_1)$, $p(e_2)$, $p(e_3)$. For the formula $\forall x. \exists y.\, x \neq y \wedge p(y)$, $\exists$ has a winning strategy: if x is e_2, pick e_3, otherwise pick e_2. The advantage of using game semantics is that we can consider how strategies in more than one structure must relate, a technique we will put to good use in Chapter 5.

2.1.3 Quantifier-Free Types

Intuitively, two tuples $a_1, \ldots, a_k \in M$, $b_1, \ldots, b_k \in N$ of elements from two structures over the same signature have the same *quantifier-free type*, or *QF-type*, if they cannot be distinguished by a quantifier-free formula. Formally, the quantifier-free type of $a_1, \ldots, a_k \in M$ is the set of quantifier-free formulas with free variables $x_1, \ldots, x_k$ that are satisfied by M and the assignment $[a_1/x_1, \ldots, a_k/x_k]$. We also define the *$b$-bounded* QF-type to only allow formulas in which all function symbols are nested at most b levels deep.

Intuitively, we can compute a representation of the b-bounded QF-type by enumerating all possible atomic formulas from the signature and $x_1, \ldots, x_k$ with maximum function nesting b, and then collecting only those that satisfy $M \cup [a_1/x_1, \ldots, a_k/x_k]$. We can ignore trivial equivalence for equality by omitting $x = x$ and keeping only one of $x = y$, $y = x$. For example, if we let M be a structure with elements e_1, e_2 and a single relation p defined by $p(e_1)$ and $\neg p(e_2)$, the QF-type is $\{p(x_1)\}$ for elements e_1, e_2, and $\{p(x_1), p(x_2), x_1 = x_2\}$ for elements e_1, e_1. In this work we will only make use of b-bounded QF-types.

2.1.4 Transition Systems

We consider a *transition system* to be a set of *states* as all structures over some signature S that satisfy an axiom Ax, a set of some initial states satisfying $Init$, a transition formula Tr which can contain both ordinary constant, predicate, and function symbols, representing values in the pre-state, and primed symbols (x') representing values in the post-state, and *safe* states satisfying the safety property $Safe$. We define *bad* states as $\neg Safe$. We define single-state implication, written $A \Rightarrow B$, and two-state implication across transitions, written $A \Rightarrow \mathrm{wp}(B)$, as:

$$A \Rightarrow B \equiv \textsc{Unsat}(A \land Ax \land \neg B) \tag{2.7}$$

$$A \Rightarrow \mathrm{wp}(B) \equiv \textsc{Unsat}(A \land Ax \land Tr \land Ax' \land \neg B') \tag{2.8}$$

where $\textsc{Unsat}(P) \equiv \neg\exists M.\, M \models P$ and P' is P with all symbols from the background signature replaced with primed versions. Note that for $A \Rightarrow \mathrm{wp}(B)$, we augment the signature with the primed symbols. Thus if a satisfying structure M exists it will have two interpretations for each original symbol: one for the pre-state and one for the post-state.

The initial states and transition relation together define *reachable* states, which are all initial states plus states which are accessible with a finite sequence of transitions from some initial state. A transition system is itself *safe* if all reachable states are safe states, or equivalently if no bad states are reachable.

Example Transition System. We present an example of a specific transition system, expressed in `mypyvy` syntax, in Figure 2.1. This simple consensus protocol, one of the simplest examples from our benchmark (Section 8.1), is a good exploration of how to model a formal system using our formulation of transition systems. We first give an overview of the operation of the algorithm, and then discuss the specifics of the syntax.

The system is modeling a simplified single-round distributed consensus protocol, with the goal allowing multiple *nodes* to decide on a unique, abstract *value*. Each node is part of some *quorum*, given by the relation `member`. The transition relation

```
sort value
sort quorum
sort node

mutable relation voted(node)
mutable relation vote(node, value)
mutable relation decided(value)
immutable relation member(node, quorum)

axiom forall Q1, Q2. exists N. member(N, Q1) & member(N, Q2)

init forall N. !voted(N)
init forall N, V. !vote(N, V)
init forall V. !decided(V)

transition cast_vote(n: node, v: value)
    modifies voted, vote
    & !voted(n)
    & (forall N, V. new(vote(N, V)) <-> vote(N, V) | N = n & V = v)
    & (forall N. new(voted(N)) <-> voted(N) | N = n)

transition decide(v: value, q: quorum)
    modifies decided
    & (forall N. member(N,q) -> vote(N,v))
    & (forall V. new(decided(V)) <-> (decided(V) | V = v))

safety forall V1, V2. decided(V1) & decided(V2) -> V1 = V2
invariant forall N, V. vote(N,V) -> voted(N)
invariant forall N, V1, V2. vote(N, V1) & vote(N, V2) -> V1 = V2
invariant forall V. decided(V) -> exists Q. forall N. member(N, Q) -> vote(N, V)
```

Figure 2.1: Example transition system, specifying the signature (**`sort`**, **`mutable`** or **`immutable relation`**), axioms (**`axiom`**), initial states (**`init`**), transitions (**`transition`**), and a safety property (**`safety`**). While not part of the transition system itself, a human-written invariant is also given.

is split into two named parts (`cast_vote` and `decide`), which are each disjuncts of the full transition relation, i.e. two states are related by the full transition relation if they are related by either transition. Each node can vote for some value, using the `cast_vote` transition. The system ensures each node can only vote once using `voted` relation. The resulting uniqueness of votes is something that needs to be proven with an invariant: while straightforward, it does not follow immediately from the definition of the system and its transitions. The `decide` transition checks that all members of a quorum have voted for a value, and then makes that value as a decision. The safety property states that at most one value is decided on. The correctness of the algorithm depends on the uniqueness of votes, the requirement for a quorum to unanimously vote before a decision, and the fact that any two quorums overlap: together, these ensure at most one value is decided on. The inference problem is to generate an invariant sufficient to prove the safety property. This example includes a human-written invariant, which is a solution to the corresponding inference problem. We discuss invariants further in Chapter 6.

Note this algorithm is very abstract. We could make it more closely reflect an implementation by e.g. splitting the existing atomic vote process into `send_msg` and `recv_msg`, with a `vote_msg()` relation to model the messages in flight. Adding these kinds of details makes the system more faithful to an implementation, but typically makes the required invariants larger and harder to infer. The flexibility of transition systems allows us to model systems other than distributed protocols: our benchmark (Section 8.1) includes a hardware cache coherency system.

Concretely, in `mypyvy` syntax we represent the transition system's signature with sort, relation, function, and constant declarations.[2] The interpretation of immutable symbols are not changed by any transitions, while mutable ones may have their interpretations changed by the action of transitions. We can use one or more axiom declarations (interpreted conjunctively to form Ax) to specify properties that all states must have. Quantified variables always quantify over a particular sort, but the sort can be inferred from context and is not written. Here the axiom constrains the quorums so that any two overlap by at least one member node. Any property that can be

[2]Constants and functions are not used in this example.

written as a first-order formula is suitable, which allows us to model objects such as total orderings that do not otherwise exist in uninterpreted logic.

The initial states and safety property are represented by one or more declarations, each interpreted conjunctively to form *Init* and *Safe*. The first `init` states that in initial states, all nodes have not voted. The transitions are a bit more involved: each transition allows arguments (e.g. `n` and `v` for `cast_vote`) that are implicitly existentially quantified. The body of the transition contains a modifies clause, which ensures that symbols not mentioned in this clause preserve their interpretations between the pre- and post-states. Within the `new(...)` syntactic form, all symbols from the system signature are primed.[3] The body can contain restrictions on the pre-state alone or relations between the pre- and post-state: in the `cast_vote` transition, the transition requires the node `n` to not have voted in the pre-state, and the voted relation in the post-state (i.e. `new(voted(...))`) is updated to hold for `n` but be unchanged for other nodes. The overall abstract transition relation for this example will be of the form $Tr = (\exists n, v.\, P) \vee (\exists v, q.\, Q)$, where P and Q are the bodies of the transitions (including restrictions from modifies clauses). Splitting the transition disjunctively turns a large Unsat query from Equation (2.8) into multiple smaller and hopefully easier queries.

2.2 Related Work

Our work draws on ideas from logic and machine learning, and is motivated by applications of quantified formulas in verification. We also discuss other techniques for invariant inference of quantified formula.

2.2.1 Logical Separability

Distinguishability of two structures with a quantified formula has been investigated through the study of Ehrenfeucht-Fraïssé (EF) games [23]. These games characterize the structures that may be separated by a formula with quantifier rank k, which

[3]Note quantified variables and argument variables are not primed within `new`.

is the maximal depth of a quantifier in the formula. EF games are traditionally only defined for a pair of structures, while we are interested in separating sets of structures. A related usage is graph isomorphism problems. Determining whether a pair of structures can be separated by *any* first-order formula is the same question as asking if they are isomorphic, which is a generalization of graph isomorphism.

2.2.2 Interpolants

Separators are related to the concept of *interpolation*, which has been applied to software and hardware verification [34, 35]. An interpolant can be viewed as a separator between sets of states described by formulas. While interpolants separate symbolic sets of states and we consider finite, concrete sets of states, the two are connected because one can be used to implement the other (i.e., by creating a formula that encodes a finite set, or in the other direction by using counterexample-guided refinement of the concrete sets).

Existing work finds quantified interpolants for alternation-free formulas [8]. Another existing interpolation technique [1] is similar to this work in that it reduces to SAT and minimizes a syntactic measure of complexity, but it considers quantifier-free interpolants over the theory of linear rational arithmetic.

2.2.3 Boolean Learnability

The problem of learning a Boolean function from examples can be seen as a separability problem where the formula is restricted to propositional logic. This problem is often investigated in models like PAC (*probably approximately correct*) learning from statistical machine learning theory, where the separator need only be probabilistically correct on the examples. In this work, we are interested only in exact separators that label all examples correctly. For example, [5] shows that in the exact learning setting, only a polynomial number of examples are required if the true separator has both a short CNF and DNF representation. In contrast, the worst case to learn an arbitrary Boolean function of n inputs requires 2^n examples, one for each input point.

Recently, [11] showed that for some classes of Boolean functions, learning inductive invariants is harder than exact learning.

2.2.4 VC dimension

The VC dimension of a class of classifier functions $\mathcal{F}$ over some domain D is the size of the largest set $C \subseteq D$ that the class can *shatter*. A class $\mathcal{F}$ shatters a set C if for any labeling of elements of C as positive and negative, a function from $\mathcal{F}$ assigns that labeling. Stated another way, the VC dimension is the largest set that can be separated by $\mathcal{F}$ in every possible way. While VC dimension was originally defined for statistical learning [44], it has been applied to study exact learning for program analysis [42].

2.2.5 Quantified Formulas in Verification

Despite their computational expense and possible undecidability, quantified formulas are used in many verification tools. In Ivy [39], Alloy [26], and Dafny [31], quantified formulas including quantifier alternation are part of the user's interface to the system. Existing invariant inference techniques based on IC3/PDR [4, 9] such as PDR$^\forall$ [27] are restricted to universally quantified invariants, and systems that would be modeled naturally with existential quantifiers must be manually transformed (if possible) to eliminate existential quantifiers.

2.2.6 Extensions of PDR/IC3.

The PDR/IC3 [4, 9] algorithm has been very influential as an invariant inference technique, first for hardware (finite state) systems and later for software (infinite state). There are multiple extensions of PDR/IC3 to infinite-state systems using SMT theories [22, 29]. [27] extended PDR/IC3 to universally quantified first-order formulas using the model-theoretic notion of *diagrams*. [18] applies PDR/IC3 to find universally quantified invariants over arrays and also to manage quantifier instantiation. Another extension of PDR/IC3 for universally quantified invariants is [32], where a quantified invariant is generalized from an invariant of a bounded, finite system. This technique of generalization from a bounded system has also been extended to quantifiers with alternations [16]. Recently, [45] suggested combining synthesis and PDR/IC3, but they focus on word-level hardware model checking and do not support quantifier alternations. Most of these works focus on quantifier-free or universally quantified invariants. In contrast, we address unique challenges that arise when supporting lemmas with quantifier alternations.

The original PDR/IC3 algorithm has also been extended with techniques that use different heuristic strategies to find more invariants by considering additional proof goals and collecting reachable states [20, 25]. Our implementation benefits from some of these heuristics, but our contribution is largely orthogonal as our focus is on inductive generalization of quantified formulas. Generating lemmas from multiple states, similar to multi-block generalization, was explored in [30].

[33] suggests a way to parallelize PDR/IC3 by combining a portfolio approach with problem partitioning and lemma sharing. Our parallelism is more fine-grained, as we parallelize the inductive generalization procedure within PDR/IC3.

2.2.7 Synthesis-Based Approaches to Invariant Inference.

Synthesis is a common approach for automating invariant inference. ICE [15] is a framework for learning inductive invariants from positive, negative, and implication constraints. Our use of separation is similar, but it is integrated into PDR/IC3's inductive generalization, so unlike ICE, we find invariants incrementally.

2.2.8 Enumeration-Based Approaches.

Another approach is to use enumerative search, for example [10], which only supports universal quantification. Enumerative search has been extended to quantifier alternations in [19], which is able to infer the invariants of complex protocols such as some Paxos variants.

Chapter 3

Quantified Separation

We give a formal definition of the quantified separation problem, both as a computational primitive useful for invariant inference, and as a decision problem. We also consider what class of formulas to allow as separators.

3.1 Definition of Separation

We take inspiration from the problem of linear separability from machine learning and statistics, where two sets of points in space, one labeled positive and the other negative, are *separable* if there is a hyperplane (i.e., a linear expression) such that the positive points are in the positive half-space and the negative points are in the negative half-space (Figure 3.1). If there is no such plane, then the points are *inseparable*.

We can define quantified separation analogously, where we replace points with first-order structures and the hyperplane by a (possibly quantified) first-order formula from some predetermined space of formulas $\mathcal{P}$. The half-spaces are replaced by satisfaction, i.e., p must be true for the positive structures and false for the negative ones. Formally:

Definition 1 (Separability)**.** *Given a signature S and set of formulas $\mathcal{P}_S$, the sets of structures A^+ and A^- defined over S are* separable *if and only if there exists a*

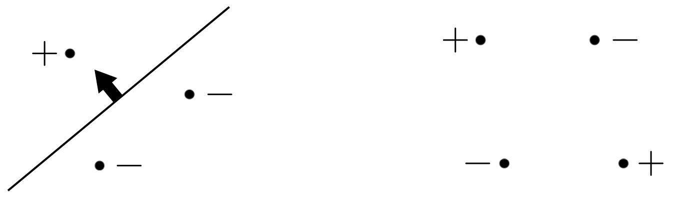

Figure 3.1: Example of two sets of points that are linearly separable and not linearly separable, respectively.

formula $p \in \mathcal{P}_S$ such that:

$$M \models p \quad \text{for } M \in A^+ \tag{3.1}$$

$$M \not\models p \quad \text{for } M \in A^- \tag{3.2}$$

If no such formula exists, we say A^+ and A^- are inseparable *or the problem is* UNSEP.

We are particularly interested in the case where $\mathcal{P}_S$ includes quantifed formulas, as this will allow us to find invariants with quantification. In particular, the algorithm we will use for separation will permit both universal and existential quantification, nested arbitrarily. This is what distinguishes our work from prior work on invariant inference. An example of such a formula with alternations was given in Figure 1.1. In that example, a quantifier alternation is necessary to separate the two sets.[1]

We can call the structures in A^+ *positive constraints*, and the structures in A^- *negative constraints*. It is useful for invariant inference to have a third kind of constraint, *implication constraints*, pairs of structures $(M, M') \in A^{\rightarrow}$, and to define an extended separation problem:

Definition 2 (Extended Separability)**.** *Given a signature S and set of formulas $\mathcal{P}_S$, sets of structures A^+, A^-,$A^{\rightarrow}$ are* separable *if and only if there exists a formula $p \in \mathcal{P}_S$*

[1]To see this, observe that a purely universal or existential formula cannot separate a structure from a sub- or superstructure (i.e. structures obtained by removing or adding elements without changing the relations between remaining or existing elements), respectively. In these sets, the positive single point is a substructure of any negative example, and the negative line segment is a substructure of the last three positive examples.

such that:

$$M \models p \qquad \textit{for } M \in A^+ \tag{3.3}$$

$$M \not\models p \qquad \textit{for } M \in A^- \tag{3.4}$$

$$(M \models p) \to (M' \models p) \qquad \textit{for } (M, M') \in A^\to \tag{3.5}$$

An implication constraint expresses the constraint that if a formula is satisfied for the first structure, it must also satisfy the second. These constraints, when the pair of structures satisfy a transition relation, are useful for generating inductive formulas. While implication constraints are important for invariant inference, they play relatively little role in either the algorithm for separation or complexity proof, so we will use Definition 1 until we discuss invariant inference in Chapter 6.

3.2 Separability as a Decision Problem

To define a decision problem, and thus allow ourselves to consider the complexity of separation, we need to fix a class of formulas to consider as separators. If we allow arbitrary first-order formulas then the problem is trivial: we can write down a formula that satisfies exactly a given structure and a disjunction of such formulas for the positive structures will separate any set, provided no structure is both positive and negative. Such formulas have at least as many quantifiers as structure elements, so it is natural to consider limiting the quantifiers in some way. We focus on the separability problem restricted to prenex normal formulas with at most k quantifers (k*-prenex*):

Definition 3. *k-SEP is the decision problem of determining whether positively and negatively labeled structures A^+, A^- are separable by a prenex formula with at most k quantifiers.*

In addition, we can define a related separation problem for when we are given a particular prefix to separate with:

Definition 4. *fixed-k-SEP is the decision problem of determining whether a given set of positively and negatively labeled structures is separable by a prenex formula with a specified quantifier prefix of size k.*

For example, Figure 1.1 is separable in 3-SEP but not in 2-SEP. Similarly, it is separable in 3-fixed-SEP for the prefix $\forall\forall\exists$ but not for the prefix $\exists\exists\exists$.

We will now explore the properties of other classes of formulas, such as those with quantifiers nested up to depth k (*k-depth*). We will show how these properties relate to those of k-prenex, and explain why some theoretical properties indicate that k-prenex might be a good choice for practical separation.

3.2.1 k-Depth Separability is in P

The quantifier-depth of a first-order formula, p, is the maximum depth of nesting of quantifiers in p. It is easy to see from known results [24] that for each fixed k, k-depth separability is in P by computing the model-theoretic *types* of the structures. The C-type, for some class of formulas C, is the set of all formulas from C true of that structure. If two structures have the same type, they are inseparable. If the types are different then there is a formula in one set and not the other that can be the separator.

Proposition 1. *For a fixed relational signature, given ℓ positively labeled and m negatively labeled structures, where each is of maximum size n and all have total size s, testing if they are k-depth separable, and if so, finding a depth k separator, can be done in time $O(s + (\ell + m)n^k)$.*

The k-dimensional Weisfeiler-Leman algorithm, known for its applications to graph isomorphism, computes a coloring of all k-tuples of vertices in an input graph. As [24] shows, this coloring of k-tuples is exactly the C^{k+1}-*type* of the k-tuple, where C^k is first-order logic with a fixed set of k variables $x_1, \dots x_k$ (which may be arbitrarily requantified) and *counting quantifiers*.

The time to compute this coloring is given as follows:

Proposition 2. *We can compute the C^k types of a given n-vertex graph in time $O(n^k \log n)$. [24]*

The algorithm in Proposition 2 works just as well, in the same running time, for L^k, first-order logic with at most k variables but no counting quantifiers. Furthermore, this algorithm works by incrementally computing the L^k or S^k type of quantifier-depth 0, 1, 2, …, until a fixed point is reached. Stopping after k rounds reduces the time needed to compute the depth k L^k type to just $O(n^k)$. In depth k, having at most k variables is no restriction. Furthermore, the same algorithm works for any relational structure with relations of arity at most k.

Corollary 1. *Given a relational structure of size s, universe size n and maximal arity at most k, we can compute its depth k type in time $O(s + n^k)$.*

Finally, to prove Proposition 1, we simultaneously compute the depth k types of the given structures. They are k-depth separable if and only if the set of types occurring in the positive structures is disjoint from the set of types occurring in the negative structures. In this case, a depth k separator is just a disjunction of the types of the positive structures. We can separate structures over signatures with function symbols as long as the nesting in the allowed separators is bounded, as fresh relations can be introduced to represent the finite set of atoms containing function symbols.[2]

These disjunctive k-depth separators encode the positive structures they separate directly, and intuitively we do not expect them to generalize well. In contrast with k-depth formulas, k-prenex formulas must share the k quantifiers amongst all the structures, so we might expect these separators, when they exist, to find some common property of the structures. One theoretical tool to analyze this intuitive overfitting behavior is by calculating the VC dimension of these classes.

3.2.2 VC Dimension of k-Depth is Exponentially Larger than k-Prenex

We analyze the VC dimension of k-depth and k-prenex formula classes over a fixed signature which has one binary relation r and no other symbols (besides equality).

[2]For example, introduce fresh relations ${r'}_1(x, y) = r(f(x), y)$, ${r'}_2(x, y) = r(f(x), g(y))$, etc. and remove function symbols. When a separator exists, these relations can be expanded to obtain a separator of the original structures.

k-Prenex A bound on the VC dimension of a class can be obtained based on the size of the class. Recall that to shatter a set of size n, we need at least 2^n different functions, so the VC dimension is bounded above by the logarithm of the size of the class. There are 2^k different prefixes, and $2k^2$ different atoms (k^2 for both the relation and equality). The number of matrices is then $2^{2^{2k^2}}$, and the overall number of formulas is $2^{k+2^{2k^2}}$. Thus the VC dimension of k-prenex formulas is at most $k + 2^{2k^2}$.

k-Depth We show that k-depth formulas may shatter a set of structures which encode a large number of disjoint graphs. Let G_k be the set of all distinct unlabeled directed graphs without self-loops on $k-1$ vertices. For a graph $g \in G_k$, we construct a graph *gadget* g' as follows: Add a new vertex v_g with a self loop, $r(v_g, v_g)$ plus edges to all the vertices in g. This new vertex prevents formulas designed to match one gadget from spuriously matching a subgraph of another gadget. Now we build a set of structures S with all $2^{|G_k|}$ patterns of presence or absence of graph gadgets for graphs $g \in G$. For each g, we can construct a formula:

$$\exists x_0, x_1, \ldots, x_{k-1}.\ r(x_0, x_0) \wedge r(x_0, x_1) \wedge \cdots \wedge r(x_0, x_{k-1})$$
$$\wedge\ (x_{i>0} \text{ are related as in } g)$$

This formula is true precisely when the gadget for g appears in the structure. To isolate a particular structure, we can conjoin the formulas for the present graphs with the negations of the formulas of all the absent ones. Then by taking the disjunction of these formulas for a set of structures, we can separate any subset of S. Thus the VC dimension of k-depth formulas is at least $2^{|G_k|}$ which is exponentially larger than $k + 2^{2k^2}$ because $|G_k|$ is exponential in k [14]. These results can be extended to other signatures as long as they have at least one non-unary relation (to encode the graph gadgets) and the function nesting depth is fixed (as in Section 3.2.1).

3.2.3 Summary of Separator Classes

In addition to the classes of separators we have already looked at, we can consider a few more related classes:

Table 3.1: Summary of important properties of separator classes. VC dimension is for the signature with a single binary relation.

Class	Sep. Complexity	VC-dim	$\wedge/\vee$-closed?
full FOL	graph-iso.	∞	✓
alternation-free	graph-iso.	∞	✓
$\forall^*/\exists^*$	NP-complete	∞	
$\forall^k/\exists^k$	P[†]	$\leq 2^{2k^2}$	
k-depth	P[†]	$\geq 2^{\lvert G_k \rvert}$	✓
k-prenex	NP-complete[†]	$\leq k + 2^{2k^2}$	

[†]for fixed k and signature.

1. Alternation-free: Formulas in which $\exists$ does not appear inside $\forall$ and vice versa.

2. $\forall^*/\exists^*$. A pair of classes, each consisting of an arbitrary number of quantifiers of one kind. These classes are closely related because swapping the labels on the structures negates the separator and switches $\forall \leftrightarrow \exists$.

3. $\forall^k/\exists^k$. A pair of classes, each with prefixes of at most k quantifiers of one kind. Each is a subclass of k-prenex formulas.

If a class of formulas is closed under $\wedge$ and $\vee$, a separator of that class exists if and only if every pair of positive and negative structures can be separated by possibly different formulas. Let Φ_{ij} separate positive structure i from negative structure j. Then the formula:

$$\bigvee_i \left(\bigwedge_j \Phi_{ij} \right)$$

is true for positive structures and false for negative structures. Thus separator classes closed under $\wedge/\vee$ are actually not desirable, because they allow a separator to be constructed from pairwise separators.

The pairwise argument allows us to give the complexity of full FOL and alternation-free separators as equivalent to graph isomorphism, because a *pair* of structures are separable with these classes if and only if they are not isomorphic. Isomorphism of first-order structures is equivalent to graph isomorphism, which is in NP but not known to be NP-complete or in P. The complexity of $\forall^*/\exists^*$ is equivalent to subgraph isomorphism because the structures are separable if and only if no negative structure is a substructure of a positive structure. Subgraph isomorphism is known to be NP-complete [6].

To summarize, for each of these classes we give the complexity of the separability decision problem, the VC dimension, and whether it is closed under $\vee$ and $\wedge$ in Table 3.1. If we adopt the common hypothesis that low VC dimension helps avoid overfitting [42], then desirable candidates are $\forall^k/\exists^k$ and k-prenex. While $\forall^k/\exists^k$ has desirable properties including low computational complexity, purely universal or existential formulas cannot express invariants of many systems, including Paxos, its variants, and even the simple consensus protocol in Figure 2.1. We thus propose k-prenex formulas as a separator class k-SEP, because it is expressive and our evaluation will show that it is often tractable in practice. One interesting property of the complexity of these classes is that the complexity of k-prenex is higher than either k-depth or $\forall^k/\exists^k$, while in terms of inclusion it sits between the two.

Now that we have justified our choice of definition for k-SEP, we will give an algorithm for separation, and then present a complexity proof that together establish k-SEP as NP-complete.

Chapter 4

Algorithm for Quantified Separation

We begin by considering separation with formulas in the quantifier-free case, and then add support for quantifiers. After this explanation, we present our separation algorithm formally, and consider important extensions to the base algorithm.

4.1 Quantifier-Free Separation

We start with an example of a signature and structure and investigate how separation can be solved in this setting. Our signature consists of a single sort A, a constant $c : A$, and two unary relations $p(x : A)$ and $q(x : A)$. Candidate separators are thus formulas such as $p(c)$, $\neg q(c)$, $\neg p(c) \vee q(c)$, etc. We can define a structure M_1 over this signature:

$$M_1 = \begin{cases} A = \{a_0, a_1\} \\ c = a_0 \\ p = \{a_0 \mapsto \top, a_1 \mapsto \bot\} \\ q = \{a_0 \mapsto \bot, a_1 \mapsto \bot\} \end{cases} \tag{4.1}$$

Now if we consider our candidate separators, we see that $M_1 \models p(c)$, $M_1 \models \neg q(c)$, and $M_1 \not\models \neg p(c) \vee q(c)$. Recall that the names of elements (a_0, a_1) cannot appear in formulas: they are not part of the signature, and only matter when evaluating atomic formulas (e.g. $p(c) \mapsto p(a_0) \mapsto \top$). Essentially, the truth of a formula is determined solely by the truth values of its atomic formulas, and so two structures that induce the same truth values to all atomic formulas must be assigned the same truth value by the separator. For M_1, we can enumerate all of the atomic formulas and their values:

$$QF(M_1) = \{c = c \mapsto \top, p(c) \mapsto \top, q(c) \mapsto \bot\} \tag{4.2}$$

This list is isomorphic to the quantifier-free type of the empty tuple (Section 2.1.3), so we will refer to such a list of atomic formula values as a *QF-type*. It is straightforward to explicitly compute a representation of the QF-type for a structure, as long as we bound the depth of function symbol nesting to ensure there are a finite number of atomic formulas.

The algorithm for quantifier-free separation is thus to compute the QF-type of each structure in A^+ and A^-. If there is any overlap between the two QF-type sets, then they are inseparable. If the sets of resulting QF-types are disjoint, then they are separable. In the separable case, we can compute an explicit separator formula. First, we can construct a formula that is true for exactly one QF-type by conjoining all atomic formulas exactly once, with negations for atomic formulas the QF-type maps to false (e.g., for $QF(M_1)$ this is $c = c \wedge p(c) \wedge \neg q(c)$). Then we can take the disjunction of all of these formulas for the QF-types in positive structures; essentially, we are constructing a formula that is true exactly for the QF-types we observe in the positive structures. Note that this construction produces extremely large formulas and is prone to overfitting; we address this overfitting in Section 4.4.

4.2 Adding Quantification

Now we assume we want to separate with a single universal, i.e., find a formula $\forall x. \varphi$, if it exists. One way of expressing the semantics of $\forall$ is to say that $M \models \forall x. \varphi$ is equivalent to $M \cup [a_i/x] \models \varphi$ for all $a_i \in A$. From the perspective of the quantifier-free matrix φ, we now have an additional symbol x to form atomic formulas, and this quantified variable is treated the same as a constant from the original signature. We

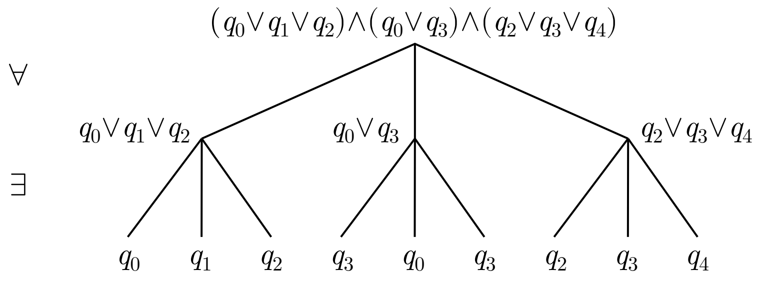
$(q_0 \lor q_1 \lor q_2) \land (q_0 \lor q_3) \land (q_2 \lor q_3 \lor q_4)$
$\forall$
$q_0 \lor q_1 \lor q_2$
$q_0 \lor q_3$
$q_2 \lor q_3 \lor q_4$
$\exists$
q_0 q_1 q_2 q_3 q_0 q_3 q_2 q_3 q_4

Figure 4.1: Example structure satisfaction formula, and the tree of assignments for a 3-element structure and the prefix $\forall\exists$. Each edge represents assigning one element to a quantified variable. The specific pattern of QF-types in the leaves is purely for illustration.

can thus reuse the reasoning about QF-types, as long as we compute the QF-types of the input structures augmented with assignments to the quantified variables:

$$QF(M_1 \cup [a_0/x]) = \{c = c \mapsto \top, p(c) \mapsto \top, q(c) \mapsto \top, p(x) \mapsto \top, q(x) \mapsto \bot\} \quad (4.3)$$
$$QF(M_1 \cup [a_1/x]) = \{c = c \mapsto \top, p(c) \mapsto \top, q(c) \mapsto \bot, p(x) \mapsto \bot, q(x) \mapsto \bot\} \quad (4.4)$$

If M_1 is a positive structure, then we require all of these QF-types to be true under φ, and if M_1 is a negative structure, we require at least one to be false. We can extend the quantifier-free algorithm to handle a single universal by computing the set of all QF-types of positive structures, and then reporting separable if for each negative structure, there is at least one QF-type that is not among the positive types.

This reasoning by checking QF-type overlap is not sufficient for more complex prefixes. If we have $\forall x.\, \exists y.$, we need for every assignment to x, some assignment to y that satisfies φ, but which QF-type is picked as true might affect the choice in other structures. Fortunately, there is already a tool for searching for these kinds of consistent assignments of truth values: SAT queries. By translating to SAT, we can ask a solver to compute which QF-types must satisfy φ.

4.3 Separation Algorithm

We first introduce Boolean variables q_i, the *QF-type variables*, which represent whether $M \cup \sigma \models \varphi$ for any M, σ where $QF(M \cup \sigma) = i$.[1] We can then recursively construct a Boolean formula for each structure M that represents whether the separator p satisfies M in terms of the q_i, using the correspondence between $\forall, \exists$ and $\wedge, \vee$. An example for the prefix $\forall\exists$ and an unspecified 3-element structure is given in Figure 4.1. Starting from the root, we assign each quantified variable to all possible elements, and going back up we construct a formula by taking the conjunction or disjunction of the children of each node depending on whether it is $\forall$ or $\exists$.

With these formulas, we can construct a query by asserting the formula directly for positive structures, and asserting the negation for negative structures. Implication constraints can be handled by asserting an implication between two formulas. To complete the algorithm, we can add an outer loop that iterates over the possible prefixes and tries to separate with each in turn.

Pseudocode for this algorithm is given in Figure 4.2. The algorithm enumerates prefixes by size up to size k, and checks whether the given structures can be separated by each one, exiting early if one is found. For each prefix, the algorithm constructs and solves a SAT query built from a conjunction of formulas derived from all structure constraints. The sat_formula function recursively constructs a Boolean formula which is true precisely when $M \models p$. The function recurses on the quantifiers in the prefix, adding all possible values of its variable to the assignment σ, and combining the sub-formulas with $\wedge$ or $\vee$. When we reach φ, the algorithm computes a b-bounded QF-type for M, σ by enumerating all finitely many well-sorted atoms a, and recording whether $M \cup \sigma \models a$.

To evaluate the complexity of this algorithm, we begin by observing the number of atoms in each QF-type is a constant given the signature, b, and k. The size of each structure formula is $O(n^k)$, where n is the number of elements in the structure.[2]

[1] Here the q_i are indexed by the full QF-type, and so i is this full object and not a number. Alternatively, we can imagine numbering the QF-types and then indexing q_i by these numbers, which is what our implementation actually does.

[2] With multiple sorts, we can take n to be the maximum size of any domain.

```
def separate(structures):
    prefix ← ""
    while size of prefix $\leq k$:
        if check_prefix(prefix, structures):
            return build_matrix(prefix, structures)
        prefix ← next_prefix(prefix)
    return $\bot$
def check_prefix(prefix, structures):
    $F$ ← {if $m$ is positive then sat_formula(prefix, [], $m$)
                    else ¬sat_formula(prefix, [], $m$)  for $m \in$ structures}
    return $\bigwedge F$ is SAT?
// For prefix $p$, assignment $\sigma$, structure $m$
def sat_formula(p, $\sigma$, m):
    if $p$ is empty:
        return SAT variable of QF-type of $\sigma$ in $m$
    ($Q\,v : S.$ rest) ← $p$
    $f$ ← {sat_formula(rest, $\sigma \cup [e/v]$, $m$)      for $e \in m$ of sort $S$}
    return if $Q = \forall$ then ($\bigwedge f$) else ($\bigvee f$)
```

Figure 4.2: Pseudocode for the separation algorithm including supporting functions.

There are $O((2m)^k)$ prefixes, where m is the number of sorts. Because the size of the queries and number of queries are both polynomial, this reduction shows that k-SEP is in NP. The remainder of this chapter discusses how to produce a concrete separator formula from this reduction, as well as practical heuristics and optimizations.

4.4 Constraining Matrices

The construction of a matrix from an assignment to q_i given in Section 4.1 will correctly separate the structures, but it produces extremely large formulas and is prone to overfitting. Instead of allowing arbitrary Boolean structure in matrices, we can limit matrices to some desired syntactic form, and add constraints to the SAT query to enforce this restriction.

For simplicity, we show how to build a matrix in conjunctive normal form, but any desired form can be used. Our matrix will consist of n clauses of literals, and we can introduce variables $y_{j,k}$ to mean literal j appears in clause k. We require that each atom appears at most once per clause, i.e. $\neg y_{j,k} \lor \neg y_{j',k}$ for the two literals j, j' of the same atom. If we let the constants $T_{i,j}$ mean that literal j is true in QF-type i, then we can construct a formula which constrains the QF-type variable to have the same truth value as the matrix in any assignment with that QF-type:

$$\begin{aligned} q_i \leftrightarrow\ & (T_{i,0} \land y_{0,0} \lor T_{i,1} \land y_{1,0} \lor \ldots) \\ & \land (T_{i,0} \land y_{0,1} \lor T_{i,1} \land y_{1,1} \lor \ldots) \\ & \land \ldots \end{aligned}$$

Because the T's are constants, this formula will simplify by effectively dropping some of the y terms in each clause. The dropped literals will be the same between clauses of q_i but a different set of literals will be dropped in the definition of some other $q_{i'}$. The variables $y_{j,k}$ directly encode the resulting separator, which can be read off directly from the assignment obtained from the SAT solver.

This process can still result in formulas that have extraneous literals, and are thus likely to be overfit. To address this issue, we perform a local minimization step after a successful SAT result by asking the solver if there is a solution with a strict subset of the literals from the last solution. As long as a smaller solution exists, we repeat this process until we find a locally minimal solution.

4.5 k-Term Pseudo-DNF

We now introduce a syntactic form for matrices that shrinks the search space relative to other forms while still allowing common invariants with quantifier alternations to be expressed. This form is inspired by the human-written invariants from our benchmark (Section 8.1), but does not depend on the specifics of invariant inference. Using this form introduces an explicit, useful bias towards formulas that are more likely to be invariants.

Conjunctive and disjunctive normal forms (CNF and DNF) are formulas that consist of a conjunction of *clauses* (CNF) or a disjunction of *cubes* (DNF), where clauses and cubes are disjunctions and conjunctions of literals, respectively: For example, $(a \lor b \lor \neg c) \land (b \lor c)$ is in CNF and $(a \land \neg c) \lor (\neg a \land b)$ is in DNF. We further define k-clause CNF and k-term DNF as formulas with at most k clauses and cubes, respectively.

We find that both CNF and DNF are not good fits for the formulas in human-written invariants. For example, consider the following formula from Paxos:

$$\begin{aligned}\forall r_1, r_2, v_1, v_2, q.\, \exists n.\, (r_1 < r_2 \land \text{proposal}(r_2, v_2) \land v_1 \neq v_2 \\ \to \text{member}(n, q) \land \text{left-round}(n, r_1) \land \neg\text{vote}(n, r_1, v_1))\end{aligned}$$

To write this in CNF, we need to distribute the antecedent over the conjunction, obtaining the 3-clause formula:

$$\begin{aligned}&(r_1 < r_2 \land \text{proposal}(r_2, v_2) \land v_1 \neq v_2 \to \text{member}(n, q)) \land \\ &(r_1 < r_2 \land \text{proposal}(r_2, v_2) \land v_1 \neq v_2 \to \text{left-round}(n, r_1)) \land \\ &(r_1 < r_2 \land \text{proposal}(r_2, v_2) \land v_1 \neq v_2 \to \neg\text{vote}(n, r_1, v_1))\end{aligned}$$

When written without $\to$, this matrix has the form $\neg a \lor \neg b \lor c \lor (d \land e \land \neg f)$, which is already in DNF. Under the k-term DNF, however, the formula requires a single-literal cube for each antecedent literal, i.e., $k = 4$. Because of the quantifier alternation, we cannot split this formula into cubes or clauses, and so a search over either CNF or DNF must consider a significantly larger search space. To solve these issues, we define a variant of DNF, k-term pseudo-DNF (k-pDNF), where one cube is negated, yielding as many individual literals as needed:

Definition 5 (k-term pseudo-DNF)**.** *A quantifier-free formula φ is in k-term pseudo-DNF for $k \geq 1$ if $\varphi \equiv \neg c_1 \lor c_2 \lor \ldots \lor c_k$, where $c_1, \ldots, c_k$ are cubes. Equivalently, φ is in k-term pDNF if there exists $n \geq 0$ such that $\varphi \equiv \ell_1 \lor \ldots \lor \ell_n \lor c_2 \lor \ldots \lor c_k$, where $\ell_1, \ldots, \ell_n$ are literals and $c_2, \ldots, c_k$ are cubes.*

Note that 1-term pDNF is equivalent to 1-clause CNF, i.e. a single clause. 2-term pDNF correspond to formulas of the form (cube) $\to$ (cube). Such formulas are sufficient for all but a handful of the lemmas required for invariants in our benchmark suite. One exception is the following formula, which has one free literal and two cubes (so it is 3-term pDNF):

$$\begin{aligned}
&\forall v_1.\ \exists n_1, n_2, n_3, v_2, v_3. \\
&\quad (d(v_1) \rightarrow \neg m(n_1) \wedge u(n_1, v_1)) \vee \\
&\quad (\neg m(n_2) \wedge \neg m(n_3) \wedge u(n_2, v_2) \wedge u(n_3, v_3) \wedge v_2 \neq v_3)
\end{aligned}$$

For a fixed k, k-clause CNF, k-term DNF, and k-term pDNF all have the same-size search space, as the SAT query inside the separation algorithm will have one indicator variable for each possible literal in each clause or cube. The advantage of pDNF is that it can express more invariant lemmas with a small k, reducing the size of the search space while still being expressive. We can also see pDNF as a compromise between CNF and DNF, and we find that pDNF is a better fit to the matrices of invariants with quantifier alternation (Section 8.3.3).

4.6 Lazy Exploration Optimization

One problem with the algorithm as described is that it always computes every assignment to quantified variables, even if not all expansions are necessary. For example, suppose we know that our matrix M satisfies $p(x) \Rightarrow M$. Then if our prefix is $\forall x \forall y \forall z$, in a positive structure once we assign x to an element e that satisfies $p(e)$, we know the formula will be true without considering the assignments of y and z. We modify the algorithm to take advantage of this fact by lazily expanding the set of assignments: every time we get a new proposed matrix that does not actually separate due to unconstrained assignments, we add the constraints which show why that matrix does not work. The SAT query starts with no restrictions and is updated incrementally until either UNSAT is produced, or a correct matrix is found. This optimization is particularly effective if the separation problem can be solved by exploring a small fraction of all assignments, which is often the case for prefixes with a large number of leading universal quantifiers.

4.7 Separating in EPR

The formulas generated by separation must eventually be processed by an SMT solver to check for a finished invariant and generate new constraints. Arbitrary quantification can cause slow queries or divergence, as first-order logic is not decidable. To mitigate this, we can use a fragment of many-sorted first-order logic, extended effectively-propositional reasoning (EPR) in which satisfiability is decidable and satisfiable formulas always have a finite model. We define EPR and then show how separation can be modified to produce only formulas in EPR.

4.7.1 Definition of EPR

Classic effectively-propositional reasoning, also known as the Bernays-Schönfinkel-Ramsey class, is a fragment of many-sorted first-order logic in which only constants and relations are permitted in the signature, and formulas have a prefix $\exists^*\forall^*$ in prenex normal form. The essence of EPR is to ensure only a finite number of ground terms can be formed. For example, if we have a function $f : s \to s$, i.e. from a sort s to itself, and a constant $x : s$, we can form an infinite number of terms $f(x), f(f(x)), f(f(f(x))), \ldots$, which can name an infinite number of elements. By eliminating functions and severely limiting quantifier alternations, the expressive power of this class is limited. We notice however, that if we let $f : s \to t$, then the infinite nesting issue disappears, as the result of f has the wrong sort to nest with itself. Therefore, we can use function symbols as long as we do not introduce a way to nest infinitely.

Formally, we define a directed graph for a formula p in negation normal form (i.e. with all negations pushed inside quantifiers):

Definition 6 (EPR)**.** *The* EPR graph *of a negation normal formula p over a signature with sorts S is a directed graph $G = (S, E)$ sorts as vertices and directed edges E where:*

1. *for every function $f : (s_1, \ldots s_n) \to s$ in the signature, then $(s_i, s) \in E$ for all $1 \leq i \leq n$*

2. *for every existential quantifier* $\exists x : s$ *in the scope of universal quantifiers* $\forall x_1 : s_1, \ldots, \forall x_n : s_n$*, then* $(s_i, s) \in E$ *for all* $1 \leq i \leq n$

A formula p is in EPR *if its EPR graph G is acyclic.*

Note that the presence of $f : s \to s$ immediately creates a self-loop in the EPR graph. Further, two functions $f : s \to t$ and $g : t \to s$ together create a cycle. The rule for existentials can be seen as the rule for functions applied to the Skolem function of the existential, e.g. the function f arising from the transformation of $\forall x.\exists y.\, \varphi$ to the equisatisfiable second-order formula $\exists f.\forall x.\, \varphi[f(x)/y]$. EPR preserves the decidability of satisfiability and finite-model property of classic EPR.

The acyclicity requirement means that EPR is not closed under conjunction, and so is best thought of as a property of the whole system, including axioms and transition relations, rather than of individual formulas. This means that ensuring acyclicity of the graph in the final system cannot be a local decision in each separation query.

4.7.2 Enforcing EPR

For invariant inference, the most straightforward way to enforce acyclicity is to decide *a priori* which edges are allowed, and to not generate separators with disallowed edges arising from quantifiers, in both the original formula and its negation.[3] In practice, enforcing EPR in separation means simply skipping prefixes that would create disallowed edges. Because separation produces prenex formulas, under this scheme some EPR formulas would be disallowed without additional effort (e.g. a prenex form of $(\forall x{:}s.\, \varphi_1) \vee (\exists y{:}s.\, \varphi_2)$ is $\forall x{:}s.\, \exists y{:}s.\, (\varphi_1) \vee (\varphi_2)$).

To enable separation to find formulas that are in EPR only in non-prenex form, we introduce an optional extension to separation that produces prenex formulas that may not be in EPR directly, but where the scope of the quantifiers can be *pushed down* into the disjunctions of the matrix to obtain an EPR formula.

We introduce extra *scope* Boolean variables $s_{i,j}$ that encode whether a particular quantified variable indexed by i appears in disjunct j, and add constraints that ensure

[3]The negation check is required as well as when checking $I \Rightarrow \mathrm{wp}(I)$, each formula will appear both positively and negatively.

the quantifiers are (1) nested consistently and (2) in such a way as to be free of disallowed edges. First, we need to encode that if a literal l containing quantified variable i is present in a disjunct j, the scope variable is true: $y_{l,j} \to s_{i,j}$. For (1), we assert for all indices i, i' in the prefix where i' is universal, and $j \neq j'$ are disjuncts, that $i < i' \wedge s_{ij} \wedge s_{ij'} \wedge s_{i'j'} \to s_{ij'}$. If we did not have this constraint, then the scope variables for a formula like $\forall x. \exists y. \forall z. \exists w.\, p(x, y, z) \vee q(z, w)$ would allow x to scope only over p, which it cannot as z covers both disjuncts. For this formula, this would be a problem if x and w have sorts that would create a disallowed EPR edge.[4] We restrict i' to universal formula because $\exists x.\, p(x) \vee q(x) \equiv (\exists x.\, p(x)) \vee (\exists x'.\, p(x'))$, and so only deeper universals can create new restrictions on the scope of outer quantifiers. If we were instead pushing down over a conjunction, only inner existentials would create scope constraints. To ensure condition (2), we need to assert for i, i' where the sorts for i, i' are part of a disallowed edge and i and i' are not the same kind of quantifier that $i < i' \wedge s_{i,j} \to \neg s_{i',j}$. We restrict both $\forall\exists$ and $\exists\forall$ so that both the formula and its negation satisfy the EPR constraint.

One final complication is that the order of quantifiers as given may not permit the most aggressive pushing down. For example, if we consider the previous example without y, $\forall x. \forall z. \exists w.\, p(x, z) \vee q(z, w)$, then we can push quantifiers down to $\forall z. (\forall x. p(x, z)) \vee (\exists w.\, q(z, w))$ by first swapping x and z, as these are both universal. Rather than enumerating all of the permutations of variables in a block as separate prefixes, we can replace $i < i'$ in the above constraints with a set of *order* Boolean variables $o_{i,i'}$, with constraints that they represent a total order over the quantified variables and only permute variables within blocks of the same type of quantifier. Thus we effectively ask the solver to decide on an order to push down the quantifiers within each block of $\forall$ or $\exists$.

After we extract a separator from the SAT assignment, we explicitly push down the quantifiers to minimal scope in the formula; the constraints we generated will ensure this is always possible and results in a formula without disallowed edges. Because this optional extension to separation makes separation queries more difficult, we only

[4]We introduce y to prevent the quantifiers for x, z from commuting and resolving the issue.

enable this mode when required, and by default use prefix skipping to implement EPR restrictions.

Chapter 5

Complexity Analysis

We now show k-SEP is NP-hard for a fixed k and signature by reduction from 3-SAT. We fix k and the signature as this puts verification of a separator in P.[1] Our reduction produces a set of structures $M_{k,\ell}$ as a function of the SAT formula, the prefix size k, and the number of universal quantifiers ℓ. For the reduction to be correct, we need to show that the resulting structures are separable if and only if the original SAT problem is satisfiable. We begin by developing intuition using the case where $k = 2$, and then we generalize to fixed-k-SEP, and then full k-SEP.

5.1 Preprocessing the SAT Problem

We first transform the 3-SAT problem into uniform-3-SAT. If any clauses are not *uniform*, i.e. contain only positive literals or only negative literals, we introduce a fresh variable x to split the clause:

$$(a \vee \neg b \vee c) \mapsto (a \vee c \vee x) \wedge (\neg x \vee \neg b)$$

[1]This does not by itself imply the problem is in NP, as the separator could be exponentially large. In Section 4.3 we showed the truth values of a separator matrix need only be specified on a polynomial number of QF-types, ensuring the size of the separator is polynomial.

m_0 +

	X	V_1
X	=	T
V_1	T	=

m_1 −

	X	V_1
X	=	F
V_1	F	=

m_2 +

	X	V_1	V_2	V_3
X	=	A	B	C
V_1	A	=	T	T
V_2	B	T	=	T
V_3	C	T	T	=

m_3 −

	X	V_1	V_2	V_3
X	=	D	E	G
V_1	D	=	F	F
V_2	E	F	=	F
V_3	G	F	F	=

Figure 5.1: Set of structures $M_{1,1}$ for a formula $(a \vee b \vee c) \wedge (\neg d \vee \neg e \vee \neg g)$. Each table specifies a structure, giving its polarity (+/−), elements (X, V_1, etc.) and for each pair of elements which binary relation holds. For example, the B in row V_2, column X in m_2 means that $B(X, V_2)$ holds in that structure, and necessarily $\neg A(X, V_2)$, $\neg T(X, V_2)$, etc.

In addition, we add fresh variables t and f along with the clauses (t), $(\neg f)$. We will use t and f to represent true and false. The resulting formula ψ is equisatisfiable with the original problem, and has only uniform clauses with at most three literals.

5.2 Example Construction for ∀∃

We first show the special case of our reduction for one of the simplest alternating prefixes, ∀∃. Our structures are defined over a signature that includes binary relations A_i, one for each $a_i \in \text{dom}(\psi)$. Note $t, f \in \text{dom}(\psi)$, so we consider T and F to be part of the A_i, but will also refer to these relations directly. All structures ensure that these relations are *anti-reflexive* ($\neg A_i(x, x)$) and *symmetric* ($A_i(x, y) \Leftrightarrow A_i(y, x)$). Further, for any two distinct elements of the same structure, exactly one of the defined relations or equality will hold between those two elements. Thus the defined relations are *total* (every non-equal pair satisfies some relation) and *mutually-exclusive* (only one relation holds for a pair).

These restrictions mean that for any assignment of x and y, either $x = y$ and no relations hold, or $x \neq y$ and for some i, exactly $A_i(x, y)$ and $A_i(y, x)$ hold. These are the only QF-types (Section 4.1) ever seen by the matrix φ of a separator: there is one QF-type for $x = y$ and one QF-type for each A_i. As observed in Section 4.1, if we have the same QF-types from different assignments to x, y, then the matrix must have the same value. This observation justifies drawing a correspondence between an assignment to Boolean variables and candidate separator formulas by equating the Boolean variable a_i to the truth value of φ on the QF-type for A_i, which by abuse of notation we denote $\varphi(A_i)$. We also let $\varphi(=)$ be the truth value of φ on the $x = y$ QF-type.

We can now describe the actual structures for a particular input:

$$\psi = (a \vee b \vee c) \wedge (\neg d \vee \neg e \vee \neg g) \wedge (t) \wedge (\neg f)$$

The construction generates an $s+1$ sized structure with elements $\{X, V_1, \ldots, V_s\}$ for a clause with s literals. The polarity of the structure matches that of the literals in the clause. If we have variables in the clause a_i, a_j, a_k, we will assert $A_i(X, V_1) \wedge A_i(V_1, X)$, $A_j(X, V_2) \wedge A_j(V_2, X)$, and $A_k(X, V_3) \wedge A_k(V_3, X)$. For positive clauses, all other distinct pairs will assert T and negative clauses will assert F. Note that we have used pairs including X to encode the Boolean variables, and those that do not include X get one of T or F (T and F represent Boolean variables in the structures for (t) and $(\neg f)$). We can represent these structures in tabular form, where the rows and columns indicate which structure element corresponds to x and y, while the cell gives the relation (or equality) that holds for that pair. As previously noted, these labels indicate the QF-type for that assignment, and if we have a matrix φ in mind then we can say the *cell itself* is true or false by applying φ. We show the structures for ψ in Figure 5.1.

With these tables in mind, we can now analyze a separator $\forall x. \exists y.\ \varphi$. In terms of the table, a $\forall\exists$ formula satisfies the structure when every row has a true cell.[2] Similarly, for a negative structure, the separator not satisfying the structure means some row is entirely false. From m_1, we conclude both $\neg\varphi(=)$ and $\neg\varphi(F)$. In m_0, we need one of $(=)$ or T to be true, but since it cannot be $(=)$, we know $\varphi(T)$.

[2]Due to symmetry, the tables in Figure 5.1 are symmetric and so we can reason about either rows or columns.

Now we inspect m_2, the structure for $(a \vee b \vee c)$. The bottom three rows all have a T, and so are trivially satisfied. The first row needs a true cell, and therefore one of $\varphi(A)$, $\varphi(B)$, or $\varphi(C)$ holds which is exactly the same constraint as the original clause. In m_3, the structure for $(\neg d \vee \neg e \vee \neg g)$, one of the rows must be entirely false. Regardless of which row is picked, one of $\varphi(D)$, $\varphi(E)$, or $\varphi(G)$ must be false and the negative clause constraint is enforced.

We now see that if a separator exists, then its matrix must produce a pattern of truth values on the QF-types which gives rise to an assignment satisfying each clause. In the other direction, we can easily construct a separator from a satisfying assignment by letting $\varphi = T(x, y) \vee A_i(x, y) \vee A_j(x, y) \dots$ where A_i is included if a_i is true in the assignment. This shows that $\forall\exists$ separation is NP-hard.

5.3 Example Construction is the Same for $\exists\forall$

A somewhat surprising fact is the same construction of structures in Figure 5.1 is also separable by the $\exists\forall$ prefix if and only if ψ is satisfiable. Under this prefix, positive structures must have an entirely true row and negative structures must have a false value in every row. When we look at m_0, m_1, this means that $(=)$ is *true* while as before T is *true* and F is *false*. We can see this change to $(=)$ interacts with the change in prefix so that m_2, m_3 still correspond to their clauses. For example, in m_3, the last three rows are trivially satisfied due to the F's, but the first row must now select one of D, E, or G to be false.

These two prefixes give us a taste of a property of our general construction: the set of structures does not depend on the exact prefix. For $k = 2$, there are only two prefixes with alternation and one set of structures. When we generalize, the structures will depend on the parameter ℓ, the number of universal quantifiers.

5.4 General Construction

We define structures $\mathcal{M}_{k,\ell}(\psi)$ over one k-ary relation symbol A_i, for each Boolean variable $a_i \in \mathrm{dom}(\psi)$. In all structures, the relations A_i are (1) symmetric, (2) anti-reflexive, (3) total, and (4) mutually-exclusive.

Each structure $M \in \mathcal{M}_{k,\ell}(\psi)$ is constructed by a function $C_{k,\ell}(c)$ of a clause $c \in \psi$. The polarity of M is that of the literals in c. Let s be the number of literals in c, so $1 \leq s \leq 3$. The domain $\mathrm{dom}(M) = X \cup Y \cup V$ will have $k - 1 + s$ elements, labeled X_i, Y_i, and V_j, with cardinality of each depending on the polarity as follows:

	$\lvert X\rvert$	$\lvert Y\rvert$	$\lvert V\rvert$
$+$	ℓ	$k-1-\ell$	s
$-$	$\ell-1$	$k-\ell$	s

There will always be exactly $k-1$ total X and Y elements, collectively the *auxiliary* elements, and one V_j element for each literal, the *variable* elements.[3] Depending on k and ℓ, it is possible that $|X| = 0$ or $|Y| = 0$. We use the order of literals in c to associate both a Boolean variable and its relation with each V_j, and we label the V_j with the j of this corresponding relation A_j. To define the interpretations of the relations, we first define our k-ary relations to be false when there are fewer than k distinct elements as arguments, ensuring anti-reflexivity. When there are k distinct arguments, we use the following function to assign a relation A_i (or one of T,F) to such sets of exactly k elements (k-sets) S:

$$R(S) = \begin{cases} A_j & \text{if } S = X \cup Y \cup \{V_j\} \\ F & \text{else if } X \subseteq S \\ T & \text{otherwise} \end{cases} \tag{5.1}$$

These rules say if a k-set contains all X and Y, and thus necessarily exactly one V_i, then the set will be assigned an A_j that matches that V_j. If they are not one of these variable k-sets, they will be F if they contain all X_i and T otherwise. The

[3]Note that the positive structure has the same number of X_i as the universal quantifiers and negative structures have the same number of Y_i as existentials.

assigned relation will be true when the elements of the k-set appear as arguments in any permutation. All other relations are false for that k-set. This construction of the relations ensures they are symmetric, total (when the arguments are a k-set), and mutually exclusive. This completes our specification of $\mathcal{M}_{k,\ell}(\psi)$.

5.5 Extracting a Satisfying Assignment from a Separator

We assume $M_{k,\ell}$ is separable by some formula Φ, which in prenex normal form is written with the given prefix of k quantifiers and ℓ universals and has matrix φ. We will construct our assignment $\mathcal{A}$ by $\mathcal{A}[a_i] = \varphi(A_i)$, and our goal is to show that this assignment satisfies ψ. We will make heavy use of the game-theoretic semantics of logic introduced in Section 2.1.2.

A first observation is that the QF-type for any assignment of k variables to distinct elements in any structure is one of the A_i. A second is that if the winning player plays with the winning strategy, the game always ends in an assignment and QF-type, on which the matrix φ assigns polarity of the structure itself. For example, in positive structures, the structure satisfying Φ implies $\exists$ plays so the matrix is always true. By carefully constructing a strategy for $\forall$, we will show the game must always end in the k-set for a variable in the clause, showing a variable from the clause must be true in $\mathcal{A}$.

First we need to show that we can force the game to end in a k-set, using the following definition and lemma:

Definition 7. *A strategy for a prenex formula is* uniqueness-preserving *if, when it plays x_{n+1} in position $(x_1, \ldots, x_n)$, distinct$(x_1, \ldots, x_n) \Rightarrow$ distinct$(x_1, \ldots, x_n, x_{n+1})$*

Another way to state this is that a strategy is uniqueness-preserving if it is never the first to play a repeated element. Winning strategies for separators of our structures must be uniqueness-preserving:[4]

[4]Note that it is always possible for a strategy to play distinct elements because our games have at most k moves and there are always at least k elements.

Lemma 1. *If $M^+, M^- \in M_{k,\ell}$, Φ is a prenex formula with k quantifiers, $M^+ \models \Phi$, and $M^- \models \neg\Phi$, then all winning strategies for $\exists$ in M^+ and $\forall$ in M^- are uniqueness-preserving.*

Proof. Consider any pair of positive and negative structures M^+ and M^-, and logical games played on both simultaneously, with $\exists$ in M^+ and $\forall$ in M^- playing with winning strategies. On each move of this combined game, the winning player plays in one sub-game and the losing player then plays in the other according to the prefix. Assume the losing player always *mirrors* the equality of the winning player: either both play new, distinct elements, or both play elements in their respective structure that repeat the element of the same prior variable. If either winning strategy is not uniqueness-preserving, then the games end with the same equalities between variables, and thus the same QF type. But a winning strategy means that this common QF-type must be false in M^- and true in M^+, which is a contradiction. □

We present a lemma which characterizes T and F, followed by lemmas that show the clause structures constrain φ like ψ constrains $\mathcal{A}$:

Lemma 2. *If $\Phi = Q_1x_1.\cdots Q_kx_k.\varphi$ is a k-prenex formula separating $M_{k,\ell}$, then $\varphi(T)$ and $\neg\varphi(F)$.*

Proof. Consider the structures $C(t)$ and $C(\neg f)$. Each has exactly k elements, so if $\forall$ (or respectively $\exists$) picks any available distinct element on its turn, then by Lemma 1, the game will end on the only k-set in each structure. But this set corresponds to T in the positive structure and to F in the negative structure, so the lemma holds. □

Lemma 3. *If Φ separates $M_{k,\ell}$ and positive clause $c \in \psi$, then $\varphi(A_a)$ holds, for some $a \in c$.*

Proof. Consider the positive structure $C(c)$. $\exists$ must have a winning strategy, and consider the $\forall$ strategy: "play any X_i if not played, otherwise any remaining unplayed element." Both of these strategies are uniqueness-preserving, so by Lemma 1 the game will end in a k-set. $\forall$ has ℓ moves, and thus all X will be played. By Equation (5.1), the resulting k-set is either a variable set for some A_a or is assigned F. By Lemma 2, it will not end in F because this would make the matrix false, and so $\varphi(A_a)$ holds. □

Lemma 4. *If Φ separates $M_{k,\ell}$ and negative clause $c \in \psi$, then $\neg\varphi(A_a)$ holds, for some $\neg a \in c$.*

Proof. Consider the negative structure $C(c)$ and the strategy for $\exists$: "play any Y_i if not played, otherwise any unplayed element." $\exists$ has $k - l$ moves, and so all Y will be played. If not all X are played by some player, then the game ends on T, which is a contradiction. Thus all of X and Y are played, and so the games ends on some variable set, say that of A_a, and thus $\neg\varphi(A_a)$. □

We can now state and prove our desired result:

Theorem 1. *If Φ separates $M_{k,\ell}$, then there exists assignment $\mathcal{A}$ which satisfies ψ.*

Proof. Let $\mathcal{A}[a_i] = \varphi(A_i)$ for all Boolean variables a_i. By Lemma 2, $\mathcal{A}[t] = \top$ and $\mathcal{A}[f] = \bot$, so clauses (t) and $(\neg f)$ are satisfied. By Lemma 3, all other positive clauses have a true variable and by Lemma 4, remaining negative clauses have a false variable. Thus $\mathcal{A}$ satisfies ψ. □

5.6 Constructing a Separator from an Assignment

We assume that ψ has satisfying assignment $\mathcal{A}$, and we are given a prefix $P = Q_0 x_0. \dots Q_{k-1} x_{k-1}$ with k quantifiers and ℓ universals. We then construct a formula $\Phi = P.\,\varphi$ that separates $M_{k,\ell}(\psi)$.

Because the prefix P is given, we need only specify the matrix φ, which will be a disjunction of cases. We add a disjunct $A(x_0, \dots, x_{k-1})$ for each Boolean variable a where $\mathcal{A}[a]$ is true. Note due to the clauses t and $\neg f$ in ψ, $T(\dots)$ will always be a disjunct and $F(\dots)$ will never be. To cover the case when two bound variables are the same element, we add a subset of the formula:

$$\begin{aligned} E_i = {} & \text{distinct}(x_0, \dots, x_{i-1}) \wedge \\ & (x_0 = x_i \vee x_1 = x_i \vee \dots \vee x_{i-1} = x_i) \end{aligned}$$

E_i expresses the fact that the variables bound before x_i are all distinct, but x_i is equal to one of them. Equivalently, E_i says that x_i is the *first* repeated bound variable. We let $E_0 \equiv \bot$ and note the E_i are mutually exclusive with each other and the relations $A_i(\ldots)$. We add E_i as a disjunct if $Q_i = \forall$, and omit E_i otherwise. This completes the specification of φ, which consists of a disjunction of some of the E_i and the relations of variables $\mathcal{A}$ assigns true.

We now show that Φ is true for positive structures in $M_{k,\ell}$ and false for negative structures. We do so by giving strategies for $\exists$ in positive structures and $\forall$ in negative structures such that the game ends with a matrix of the correct polarity.

Lemma 5. *$C(c) \models \Phi$ for a positive structure $C(c) \in M_{k,\ell}$*

Proof. We know $\mathcal{A}$ satisfies ψ, so one of the variables $a \in c$ satisfies $\mathcal{A}[a]$ and thus $A(\ldots) \in \varphi$. To show Φ is true we make the game end in A or T. Let the element of $C(c)$ which represents a be V_a. Our $\exists$ strategy will be: "play the first unplayed element from V_a, any Y, or any X, in that order." Note that our strategy is uniqueness-preserving. If $\forall$ plays a repeated element, then the corresponding E_i in φ is true regardless of the rest of the played elements, and the matrix is true.

If all played elements are distinct, then note as $\exists$ with $k - \ell$ moves we have ensured that V_a and all Y are played. If all X are played, then we end on A. If not, we end on T. Both relations are in φ, and so the matrix is true. Thus, we can always force the game to end with a true matrix, and so our strategy is winning for $\exists$ and the formula is true. $\square$

Lemma 6. *$C(c) \not\models \Phi$ for a negative structure $C(c) \in M_{k,\ell}$*

Proof. $\mathcal{A}$ assigns one of the variables $\neg a \in c$ false, so Φ is false if as $\forall$ we can force the game to end in A or F, because neither will be in φ. Now if $\exists$ plays a repeated element then one of the E_i will be true, but now this time that E_i is omitted from φ. But because all E_i and A are mutually exclusive, no disjunct will be true and the matrix is false regardless of future moves. Now we give a strategy for $\forall$ as: "play the first unplayed of V_a, any X, or any Y, in that order." Now our strategy ensures, due to our ℓ moves, that all of X and V_a will be played. Thus the game ends either in A or F, both of which are absent from φ, and so the matrix will always be false and thus the formula is not true. $\square$

Theorem 2. *If ψ has satisfying assignment $\mathcal{A}$, Φ as defined above separates $M_{k,\ell}(\psi)$*

Proof. Follows from Lemmas 5 and 6. □

5.7 Extending to k-SEP

We have shown that given a particular prefix, we can reduce SAT to separability. However, this does not immediately imply that k-SEP is also difficult, because now the prefix is not fixed, and the extra flexibility might make the problem easier. We extend our construction slightly by adding structures for the trivially satisfied clauses $(t \vee f)$ and $(\neg t \vee \neg f)$. Then we can prove the following lemmas.

Lemma 7. *If Φ is a prenex formula with at most k quantifiers and Φ separates $M_{k,\ell}$, then Φ has exactly k quantifiers.*

Proof. Assume for sake of contradiction that Φ has fewer than k quantifiers. Then it does not have k distinct terms, so all defined relations are false in the matrix φ. Thus the matrix φ is logically equivalent to some formula which only uses equality, and such a formula cannot distinguish two structures of opposite polarity but equal cardinality, such as $C(t)$ and $C(\neg f)$ from our construction. Thus, Φ cannot be a separator and we have a contradiction. □

Lemma 8. *If Φ is a k-prenex formula separating $M_{k,\ell}$, then Φ must have ℓ universals.*

Proof. Assume for sake of contradiction Φ has $u > \ell$ universals. Consider the positive structure $C(t \vee f)$ and $\forall$ strategy "pick V_f if available, then any X_i, then any Y_i." $\forall$ has at least $\ell + 1$ moves, and so the element V_f as well as all X will be picked. If all $k-1$ auxiliary elements are picked, then the variable set corresponding to F will have been picked. If not all $k-1$ auxiliary elements are picked, then because all X_i were, the k-set will still be that of F. Thus the game will always end in F, which is a contradiction.

Assume for sake of contradiction Φ has $u < \ell$ universals. Consider the negative structure $C(\neg t \vee \neg f)$ and $\exists$ strategy "pick V_t if available, then any Y_i, and finally X_i." Because this is a negative structure, $\exists$ has $k - u \geq k - \ell + 1$ moves and element V_t

and all Y are picked. If all the X are picked by some player then the game ends in the variable set for T. Otherwise one of the X is missing, and so the k-set is assigned T. The game always ending in T is a contradiction for a negative structure. $\square$

Together, these lemmas establish that any separator must have the same number of universal and existential quantifiers as suggested by k and ℓ. Note the only assumption about the prefix in Lemmas 3 and 4 was that there were ℓ universals and $k - \ell$ existentials, which we have now established. This means that Theorem 1 is also true for k-SEP. Note that the separator Φ can have any prefix with the right number of existentials. For the other direction, we do not need to modify any reasoning because the construction of a formula already made no assumptions about the prefix, and we can use that reasoning to construct separators with any permutation of quantifiers in the prefix.

5.8 Summary of Complexity Results

With Theorems 1 and 2, we have now shown there is a construction $M_{k,\ell}(\psi)$ that shows fixed-k-SEP and k-SEP are both NP-hard. In Section 4.3 we gave an algorithm for solving k-SEP with an oracle for SAT, which shows that k-SEP is in NP, and together these show separation is NP-complete.

Chapter 6

Invariant Inference

With the definition and algorithm for separation in hand, we now consider the problem of invariant inference. We first define the problem, and then discuss two algorithms for invariant inference, including how they can be adapted to use separation. We defer a complete description of our algorithm to Chapter 7.

6.1 Definition of Invariant Inference

A central question in the design and verification of systems is whether they are safe, i.e., whether some safety property is satisfied by all the reachable states of the system. Further, simply knowing whether the system is safe or not is insufficient: we desire an independently checkable proof of safety. For the transition systems (Section 2.1.4) we are interested in, we represent this proof in the form of an *inductive invariant establishing safety* for the given transition system.

A formula is *invariant* for a transition system if it is satisfied by all reachable states. Invariantness is not a very useful notion because a formula may be invariant but not provable within a given logic. Therefore, we use the weaker notion of *inductiveness*. A formula p is inductive if it is satisfied by all initial states and preserved by all transition edges, i.e. if $p \Rightarrow \mathrm{wp}(p)$. An inductive formula is necessarily invariant by induction over the number of transition steps, but the reverse is not necessarily true. Finally, a formula *establishes safety* if it implies *Safe*. These conditions can be summarized as:

Definition 8 (Invariant)**.** *An* inductive invariant establishing safety *is a formula I satisfying:*

$$\mathit{Init} \Rightarrow I \tag{6.1}$$

$$I \Rightarrow \mathrm{wp}(I) \tag{6.2}$$

$$I \Rightarrow \mathit{Safe} \tag{6.3}$$

For brevity, we use the term invariant *for such formulas.*[1]

Together, Equations (6.1) and (6.2) mean that I is *inductive* for the system, and Equation (6.3) ensures I establishes safety. We can now define invariant inference itself:

Definition 9 (Invariant Inference)**.** *Given a transition system (including a safety property),* invariant inference *is the problem of computing a formula I satisfying Equations* (6.1) *to* (6.3)*, if it exists. An invariant inference algorithm is* sound *if it only returns formulas satisfying Equations* (6.1) *to* (6.3)*, and it is* complete *if whenever such an I exists, it returns such a formula without diverging.*

We are interested in sound but not necessarily complete algorithms for invariant inference. Further, we are only interested in invariant inference for safe systems, where a suitable I exists. Although our implementation supports detecting and reporting unsafe systems, existing techniques such as model checkers are more suitable for determining if systems are unsafe. We expect these other techniques to be used in concert with invariant inference, which means the focus of inference can be exclusively on safe systems. Although we do not study unsafe systems formally, our algorithm (Section 6.3) can opportunistically prove systems are unsafe by showing a bad state is reachable.

For concreteness, we make the assumption that *Safe* and I can be broken into conjuncts, i.e. $\mathit{Safe} = \mathit{Safe}_0 \wedge \mathit{Safe}_1 \wedge \ldots$ and $I = I_0 \wedge I_1 \wedge \ldots$. We overload terminology to refer to the conjuncts of *Safe* as safety properties, and refer to the conjuncts of I as

[1]We do not have any use for the original notion of an invariant formula, so this overloading is not ambiguous.

invariants. We also assume that the conjuncts of I are a superset of those in *Safe*, e.g. $I_0 = \mathit{Safe}_0$, etc. This assumption trivially ensures $I \Rightarrow \mathit{Safe}$. The invariant inference problem can then be seen as the problem of generating zero or more invariants such that the resulting I as a whole is an inductive invariant establishing safety.

6.2 ICE Learning with Separators

Possibly the simplest invariant inference algorithm using separators is *ICE learning* [15]. The algorithm works by refining a candidate invariant I by incrementally adding constraints derived from counterexamples to equations Equations (6.1) to (6.3). Positive counterexamples arise from a first-order SAT query of the negation of Equation (6.1), and negative examples likewise come from Equation (6.3). Equation (6.2) is interesting because as observed in [41, 15], counterexamples are exactly implication constraints: I is only required to be true on the post-state if it is true for the pre-state. These SAT queries can be discharged by a standard SMT solver with support for quantified formulas. After a new constraint is added, the separation procedure produces a new I satisfying all known constraints. When no counterexamples exist, the candidate I satisfies Equations (6.1) to (6.3) and the algorithm succeeds.

This algorithm requires learning the invariant monolithically — it is not possible to learn conjuncts of a larger invariant piece by piece. Nevertheless, this algorithm is able to correctly infer the invariant for a few of the the smallest examples from our evaluation (Section 8.3, Table 8.2), including ring-id and firewall. The firewall example is notable for requiring an invariant with quantifier alternation while still being solvable with ICE learning. To avoid this monolithic learning requirement, we turn to a significantly more complex invariant inference algorithm that learns formulas incrementally.

6.3 PDR/IC3

PDR/IC3 is an invariant inference algorithm first developed for finite state model checking [4] and later extended to various classes of infinite-state systems. We

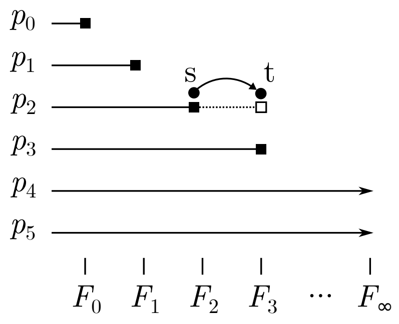

Figure 6.1: Illustration of lemmas (p_i) and frames (F_i) in PDR/IC3. Each lemma is recorded as being in some finite frame (■) or in F_∞ ($\rightarrow$). In this example, p_2 cannot be pushed to F_3 because of the transition $s \to t$, where $s \models F_2$ and $t \not\models p_2$.

describe PDR/IC3 as in [25]. PDR/IC3 maintains *frames* F_i as conjunctions of formulas (*lemmas*) representing overapproximations of the states reachable in at most i transitions from *Init*. Finite frames ($i = 0, \ldots, n$) and the frame at infinity ($i = \infty$) satisfy:

$$Init \Rightarrow F_0 \tag{6.4}$$

$$F_i \Rightarrow F_{i+1} \tag{6.5}$$

$$F_n \Rightarrow F_\infty \tag{6.6}$$

$$F_i \Rightarrow \mathrm{wp}(F_{i+1}) \tag{6.7}$$

$$F_\infty \Rightarrow \mathrm{wp}(F_\infty) \tag{6.8}$$

Equations (6.4) to (6.6) mean *Init* $\Rightarrow F_i$ for all i. We ensure Equations (6.5) and (6.6) by restricting frames to subsets of the prior frame, when taken as sets of lemmas. Equations (6.7) and (6.8) say each frame is *relatively inductive* to the prior frame, except F_∞ which is relatively inductive to itself and thus inductive for the system. To initialize, the algorithm adds the (conjuncts of) *Init* and *Safe* as lemmas to F_0. The algorithm then proceeds by adding lemmas to frames using either *pushing* or *inductive generalization* while respecting this meta-invariant, gradually tightening the bounds on reachability until $F_\infty \Rightarrow$ *Safe*. We can push a lemma $p \in F_i$ to F_{i+1}, provided $F_i \Rightarrow \mathrm{wp}(p)$. When a formula is pushed, the stronger F_{i+1} may permit us to push one or more other formulas, possibly recursively, and so we always push all possible lemmas until a fixpoint is reached. Any mutually relatively inductive set of lemmas does not have a finite fixpoint, and we detect such sets (by checking whether $F_i = F_{i+1}$) and add these lemmas to F_∞.

If the algorithm cannot push a lemma p_a, there is a model of $\neg(F_i \Rightarrow \text{wp}(p_a))$, which is a transition $s \to t$ where $s \in F_i$ and $t \not\models p_a$. If s has no predecessors in F_{i-1} (or $i = 0$), then we say s is the *proof obligation* for p_a. If s does have predecessors in the prior frame F_{i-1}, we recursively follow the predecessor relation until either we find a state s' with no predecessors, or we get to F_0. Then s' becomes the proof obligation for p_a. We keep track of which frame (labeled F_j) we found s' in. If s' is in F_0, the algorithm has discovered a chain of transitions showing that p_a violates a reachable state. If p_a is a safety property, then this show the system is unsafe. If p_a is any other lemma, it shows that p_a cannot be part of an inductive invariant.

To generate new lemmas, we *block* the proof obligation s in F_i (or s' in F_j) by using an inductive generalization (IG) query to *learn* a new lemma that eliminates s. An IG query finds a formula p satisfying:

$$s \not\models p \tag{6.9}$$

$$\mathit{init} \Rightarrow p \tag{6.10}$$

$$F_{i-1} \land p \Rightarrow \text{wp}(p) \tag{6.11}$$

If we can learn such a lemma p, it can be added to F_i and all previous frames, and removes at least the state s stopping p_a from being pushed. With the addition of p, p_a may or may not be immediately pushed, as it may have other proof obligations. Classic PDR/IC3 always chooses to block the proof obligation of a safety property, but other strategies have been considered (Section 7.4.1, [25]).

The technique used to solve the IG query controls what kind of invariants we are able to discover (e.g. universal formulas as in [27]). In this work we use separation to solve for p, which lets us infer invariants with quantifier alternations. PDR/IC3 has the advantage of learning the invariant incrementally: in general, each IG query will learn a lemma which may become one conjunct of the final invariant. This incrementality means that we can find invariants with dozens of conjuncts, but each separation query can remain much smaller in scope.

Chapter 7

Adapting PDR/IC3 with Quantified Separation

We now present our PDR/IC3-based invariant inference algorithm. We start with a discussion of inductive generalization, followed by several optimizations we made at the level of the overall algorithm. Finally, we put these pieces together to give a full description of our algorithm. At a high level, our algorithm is composed of three nested refinement loops, as shown in Figure 7.1. We have previously discussed the innermost loop in presenting the separation algorithm, and we now turn to the middle and outer loops.

7.1 Naive Inductive Generalization with Separation

We first discuss naively solving an IG query with separation (as in [28]), and then discuss the problems with this strategy. An IG query can be solved with separation by a simple iterative refinement loop. This loop performs a series of separation queries with an incrementally growing set of structure constraints. Starting with a negative constraint s for the state to block, we ask for a separator p and check if Equations (6.10) and (6.11) hold for p using a standard SMT solver. If both hold, p is a solution to the

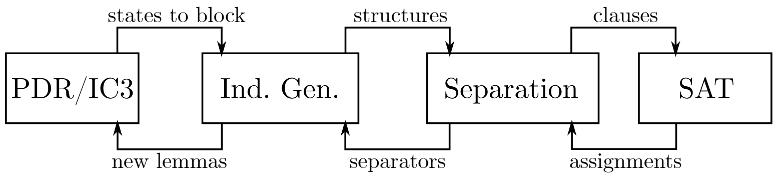

Figure 7.1: The three refinement loops of our algorithm. The PDR/IC3 provides states to block in a particular frame, which are converted into structure constraints, which become clauses in a SAT query. In the reverse direction, SAT assignments become separators, which become new lemmas in a PDR/IC3 frame. Each loop continues until a correct solution is found or the constraints are unsatisfiable.

IG query. Otherwise, the SMT solver produces a model that becomes either a positive constraint (corresponding to an initial state p violates) or an implication constraint (a transition edge that shows p is not relatively inductive to F_{i-1}), respectively.

We observe a few key points about the algorithm for separation: (i) separation considers each potential quantifier prefix essentially independently, (ii) complex IG queries can result in hundreds or thousands of constraints, and (iii) prefixes, as partitions of the space of possible separators, vary greatly in how quickly they can be explored. Further, with the naive approach where the prefixes are considered internally by the separation algorithm, even if the separation algorithm uses internal parallelism, there is still a serialization step when a new constraint is required. As a consequence of (ii) and (iii), a significant failure mode of this naive approach is that the search becomes stuck generating more and more constraints for difficult parts of the search space that ultimately do not contain an easy-to-discover solution to the IG query.

7.2 Prefix Search at the Inductive Generalization Level

To fix the problems with the naive approach, we lift the choice of prefix to the IG level, partitioning a single large separation query into a query for each prefix (e.g. use fixed-k-SEP instead of k-SEP). Each sub-query can be explored in parallel, and each can proceed independently by querying for new constraints (using Equations (6.10) and (6.11) as before) without serializing by waiting for other prefixes. We call this new technique a *breadth-first* search, because the algorithm can spend approximately equal time on many parts of the search space, instead of a *depth-first* search which exhausts all possibilities in one region before moving on to the next. When regions have greatly varying times to search, the breadth-first approach prevents expensive regions from blocking the search in cheaper regions. This improvement relies on changing the division between separation and inductive generalization: without the knowledge of the formulas (Equations (6.10) and (6.11)) that generate constraints, the separation algorithm cannot generate new constraints on its own.

A complicating factor is that in addition to prefixes varying in difficulty, sometimes there are entire classes of prefixes that are difficult. For example, many IG queries have desirable universal-only solutions, but spend a long time searching for separators with alternations, as there are far more distinct prefixes with alternations than those with only universals. To address this problem, we define possibly overlapping sets of prefixes, called *prefix categories*, and ensure the algorithm spends approximately equal time searching for solutions in each category (e.g., universally quantified invariants, invariants with at most one alternation and at most one repeated sort). Within each category, we order prefixes to further bias towards likely solutions: first by smallest quantifier depth, then fewest alternations, then those that start with a universal, and finally by smallest number of existentials.

7.3 Algorithm for Inductive Generalization

We present our algorithm for IG using separation in Figure 7.2. Our algorithm has a fixed number N of worker threads which take prefixes from a queue subject to prefix restrictions, and perform a separation query with that prefix. Each worker thread calls next-prefix() to obtain the next prefix to consider, according to the order discussed in the previous section. To solve a prefix P, a worker performs a refinement loop as in the naive algorithm, building a set of constraints $C(P)$ until a solution to the IG query is discovered or separation reports UNSEP.

While we take steps to make SMT queries for new constraints as fast as possible (Section 7.4.4), these queries are still expensive and we thus want to re-use constraints

```
def IG($s$: state, $i$: frame):
    $\forall P.\ C(P) = \{\text{Negative}(s)\}$;
    for $i = 1 \ldots N$ in parallel:
        while true:
            $P$ = next-prefix();
            while true:
                $p$ = separate $C(P)$;
                if $p$ is UNSEP:
                    break
                elif any $c \in R_C(P)$ and $p \not\models c$:
                    add $c$ to $C(P)$
                elif ($c$ := SMT check Equations (6.10) and (6.11)) $\neq$ UNSAT:
                    add $c$ to $C(P)$
                else:
                    return $p$ as solution
```

Figure 7.2: Pseudocode for our proposed inductive generalization algorithm.

between prefixes where it is beneficial. Re-using every constraint discovered so far is not a good strategy as the cost of checking upwards of hundreds of constraints for every candidate separator is not justified by how frequently they actually constrain the search. Instead, we track a set of *related constraints* $R_C(P)$ for a prefix P. We define related constraints in terms of *immediate sub-prefixes* of P, written $S(P)$, which are prefixes obtained by dropping exactly one quantifier from P, i.e. the quantifiers of $P' \in S(P)$ are a subsequence of those in P with one missing. We then define $R_C(P) = \cup_{P' \in S(P)} C(P')$, i.e. the related constraints of P are all those used by immediate sub-prefixes. While $S(P)$ considers only immediate sub-prefixes, constraints may propagate from non-immediate sub-prefixes as the algorithm progresses.

Constraints from sub-prefixes are used because the possible separators for those queries are also possible separators for the larger prefix. Thus the set of constraints from sub-prefixes will definitely eliminate some potential separators, and in the usual case where the sub-prefixes have converged to UNSEP, will rule out an entire section of the search space. We also opportunistically make use of known constraints for the same prefix generated in prior IG queries, as long as those constraints still satisfy the current frame.

Overall, the algorithm in Figure 7.2 uses parallelism across prefixes to generate independent separation queries in a breadth-first way, while carefully sharing only useful constraints. From the perspective of the global search for an inductive invariant the algorithm introduces two forms of inductive bias: (i) explicit bias arising from controlling the order and form of prefixes (Section 7.2), and (ii) implicit bias towards formulas which are easy to discover.

7.4 An Algorithm for Invariant Inference

We now take a step back to consider the high-level PDR/IC3 structure of our algorithm. As in all PDR/IC3 variants, we use IG to block backward reachable states (i.e., must-proof obligations obtained from attempting to push a safety property). We next discuss blocking states that are not backward reachable from a bad state as a heuristic for finding additional useful lemmas, and a heuristic that blocks multiple states with one lemma. We then discuss the impact of the EPR logic fragment and another technique to increase the robustness of SMT solvers. Finally, we tie everything together to give a complete description of our proposed algorithm.

7.4.1 May-proof obligations

In classic PDR/IC3, the choice of proof obligation to block is always that of a safety property. [25] proposed a heuristic that in our terminology is to block the proof obligation of other existing lemmas, under the heuristic assumption that current lemmas in lower frames are part of the final invariant but lack a supporting lemma to make them inductive. The classic blocked states are known as *must-proof obligations*, as they are states that must be eliminated at some point to prove safety. In contrast, these heuristic states are *may-proof obligations*, as they may or may not be necessary to find a solution. Our algorithm selects these lemmas at random, biased towards lemmas with smaller matrices and in later frames.

To compute a proof obligation, we recursively find any predecessors in the prior frame, if they exist. For may-proof obligations, this recursion can potentially reach all the way to an initial state in F_0,[1] and thus proves that the entire chain of states is reachable—i.e., the states cannot be blocked. This fact shows that the original lemma is not part of any final invariant and cannot be pushed past its current frame; it also provides a positive structure constraint useful for future IG queries.

7.4.2 Multi-block Generalization

After an IG query blocking state s is successful, the resulting lemma p may cause the original lemma that created s to be pushed to the next frame. If not, there will be a new proof obligation s'. If s' is in the same frame, we can ask whether there is a single IG solution formula p_1 which blocks both s and s'. If we can find such a p_1, it is more likely to generalize past s and s', and we should prefer p_1. This search for p_1 is straightforward to do with separation: we incrementally add another negative constraint to the existing separation queries. To implement *multi-block generalization*, we continue an IG query if the new proof obligation is suitable (i.e., exists and is in the same frame), accumulating as many negative constraints as we can until we do not have a suitable state or we have spent as much time as the original query. This timeout guarantees we do not spend more than half of our time on generalization, and protects us in the case that the new set of states cannot be blocked together with a simple formula.

7.4.3 Enforcing EPR

As discussed in Section 4.7, EPR is a useful logic fragment due to its decidability and finite model property. EPR requires acyclicity of a graph derived from both the functions in the signature and Skolem functions from quantifiers. For invariant inference with PDR/IC3, the most straightforward way to enforce acyclicity is to decide *a priori* which edges are allowed, and to not infer lemmas with disallowed Skolem edges. Without this fixed set of allowed edges, adding a lemma to a frame may prevent a necessary lemma from being added to the frame in a later iteration, as PDR/IC3 lacks a way to remove lemmas from frames. Requiring the set of allowed

[1]For unsafe transition systems, recursing to F_0 can also occur for must-proof obligations.

edges as input is a limitation of our technique and other state-of-the-art approaches (e.g. [19]). We do note that not all algorithms admit invariants in EPR, and manual translation of the transition system itself can be required to allow an EPR invariant. We hope that future work expands the scope of decidable logic fragments, so that systems require less effort to model in such a fragment. It is also possible that our algorithm could be wrapped in an outer search over the possible acyclic sets of edges.

7.4.4 SMT Robustness

Even with EPR restrictions, some SMT queries we generate are difficult for the SMT solvers we use (Z3 [7] and CVC5[2]), sometimes taking minutes, hours, or longer. This wide variation of solving times is significant because separation, and thus IG queries, cannot make progress without a new structure constraint. We adopt several strategies to increase robustness: periodic restarts, running multiple instances of both solvers in parallel, and *incremental queries*. Our incremental queries send the formulas to the SMT solver one at a time, asserting a subset of the input. An UNSAT result from a subset can be returned immediately, and a SAT result can be returned if there is no un-asserted formula the model violates. Otherwise, one of the violated formulas is asserted, and the process repeats. This process usually avoids asserting all the discovered lemmas from a frame, which significantly speeds up many of the most difficult queries, especially those with dozens of lemmas in a frame or those not in EPR.

7.4.5 Complete Algorithm

Figure 7.3 presents the pseudocode for our algorithm, which consists of two parallel tasks (learning and heuristic), each using half of the available parallelism to discharge IG queries, and pushing to fixpoint after adding any lemmas. In this listing, the obligation(ℓ) function computes the state and frame to perform an IG query in order to push ℓ, and the push() function pushes all lemmas until a fixpoint is reached, and marks the invariant as found if all safety properties reach F_∞. The heuristic task

[2]Successor to CVC4 [2].

additionally may find reachable states, and thus mark lemmas as bad. We cancel an IG query when it is solved by a lemma learned or pushed by another task.

Our algorithm is parameterized by the logic used for inductive generalization, and thus the form of the invariant. We support universal, EPR, and full first-order logic (FOL) modes. Universal mode restricts the matrices to clauses, and considers predecessors of superstructures when computing obligation() (as in [27]). EPR mode takes as input the set of allowed edges and whether to enable EPR pushdown (Section 4.7). In FOL mode, there are no restrictions on separation prefixes.

```
def P-Fol-Ic3():
    $F_0 = \text{Init} \cup \text{safety}$;
    push();
    start Learning(), Heuristic();
    wait for invariant;
def Multiblock($\ell$: lemma, $s$: state, $i$):
    $S = \{s\}$;
    while not timed out:
        $p = \text{IG}(S, i)$;
        speculatively add $p$ to frame $i$;
        $s', i' = \text{obligation}(\ell)$;
        remove $p$ from frame $i$;
        if $i = i'$:
            add $s'$ to $S$;
        else:
            break
    add $p$ to frame $i$;
    push();
def Learning():
    while true:
        $s, i = \text{obligation(safety)}$;
        Multiblock(safety, $s$, $i$);
def Heuristic():
    while true:
        $\ell$ = random lemma before safety;
        $s, i = \text{obligation}(\ell)$;
        if $i = 0$:
            mark $s$ reachable;
            mark bad lemmas;
        else:
            Multiblock($\ell$, $s$, $i$);
```

Figure 7.3: Pseudocode for our proposed inference algorithm, P-Fol-Ic3.

Chapter 8

Evaluation

We evaluate our separation algorithm and PDR/IC3-based invariant inference algorithm on a benchmark of distributed protocols taken from prior work. We first describe the benchmark, including exploring various properties of the human-written invariants of each example. We then evaluate the separation algorithm by asking whether it can reproduce each invariant conjunct solely from positive and negative structures. Finally, we evaluate our invariant inference algorithm on our benchmark, compare our algorithm to other techniques, and present an ablation study that investigates the impact of several significant extensions to separation and PDR/IC3.

8.1 Invariant Inference Benchmark

Our benchmark is composed of invariant inference problems from prior work on distributed protocols [39, 38, 12, 37, 43, 3, 13], written in or translated to the `mypyvy` language [36]. Our benchmark contains a total of 30 problems, ranging from simple (toy-consensus-forall, firewall) to complex (stoppable-paxos-epr, bosco-3t-safety). All our examples are safe transition systems with a known, human-written invariant.

We give the names of the examples and some summary statistics in Table 8.1. Some problems admit invariants that are purely universal, and others use universal and existential quantifiers, with some of these in EPR. We also give the number of conjuncts of the human-written invariant ($|I|$), the maximum number of quantifiers in any individual conjunct (Quants.), the number of sorts in the signature (Sorts), and the total number of constant, relation, and function symbols in the signature (Symbols). The complexity of the examples, as measured by these statistics, varies considerably between the various examples. Most Paxos variants include both an -epr version and a -forall version, where the former requires an invariant with quantifier alternations, and the latter has been augmented with ghost state such that it can be proven with a universally quantified invariant.[1]

For the formula learning experiment, we used the existing division of the invariant, by the author of each example, into a number of conjuncts. Each conjunct becomes a golden formula G. The golden formulas are not necessarily in prenex normal form and have anywhere from no quantifiers to 7 quantifiers. The decomposition is not necessarily minimal; some conjuncts could themselves be divided further into equivalent conjuncts. We had a total of 268 formulas, and a histogram of formula counts by the number of quantifiers can be seen in the total heights of bars in Figure 8.1.

8.2 Learning Formulas

To evaluate separation independently of any particular invariant inference procedure, we use a process that *learns* some golden formula G from labeled structures. We start with an empty set of structure constraints, and ask for a separator p. Then we ask whether $(p \Rightarrow G) \wedge (G \Rightarrow p)$, by querying whether its negation is satisfiable using CVC5 [2] and Z3 [7], SMT solvers that support quantifiers and producing models. If the query is UNSAT, then p and G are equivalent, and we have learned G. Otherwise, the solver returns a model M of the query, which is a structure on which p and G differ. We add M to our set of structure constraints labeled according to whether $M \models G$ or $M \not\models G$, and repeat with a new candidate p. Unlike the IG algorithm given in Section 7.3, this process is sequential, except for the parallelism between SMT solvers in a single query as described in Section 7.4.4.

[1]This augmentation is performed by hand.

Table 8.1: Statistics of invariant inference benchmark, including the logic of the example (∀ for universal, ✓ for EPR, − for FOL), the number of conjuncts of the human-written invariant ($|I|$), the maximum number of quantifiers in any individual conjunct (Quants.), the number of sorts in the signature (Sorts), and the number of symbols (Symbols).

Example	Epr	$\|I\|$	Quants.	Sorts	Symbols
lockserv	∀	8	2	1	5
toy-consensus-forall	∀	3	3	3	5
ring-id	∀	3	3	2	5
sharded-kv	∀	4	5	3	3
ticket	∀	13	3	2	8
learning-switch	∀	2	4	1	2
consensus-wo-decide	∀	4	3	2	7
consensus-forall	∀	6	3	3	8
cache	∀	36	6	3	13
paxos-forall	∀	6	5	4	11
flexible-paxos-forall	∀	6	5	5	12
stoppable-paxos-forall	∀	16	6	6	17
fast-paxos-forall	∀	13	5	5	16
vertical-paxos-forall	∀	13	6	5	18
firewall	−	1	2	1	3
sharded-kv-no-lost-keys	✓	1	4	3	3
toy-consensus-epr	✓	3	3	3	4
ring-id-not-dead	−	4	3	2	6
consensus-epr	✓	6	3	3	7
client-server-ae	✓	1	3	3	4
client-server-db-ae	−	4	3	4	7
hybrid-reliable-broadcast	−	7	4	3	11
paxos-epr	✓	5	6	4	9
flexible-paxos-epr	✓	5	6	5	10
multi-paxos-epr	✓	7	6	6	13
stoppable-paxos-epr	✓	16	6	6	15
fast-paxos-epr	✓	11	6	5	12
vertical-paxos-epr	✓	10	7	5	15
block-cache-async	−	44	5	4	22
bosco-3t-safety	✓	10	6	5	13

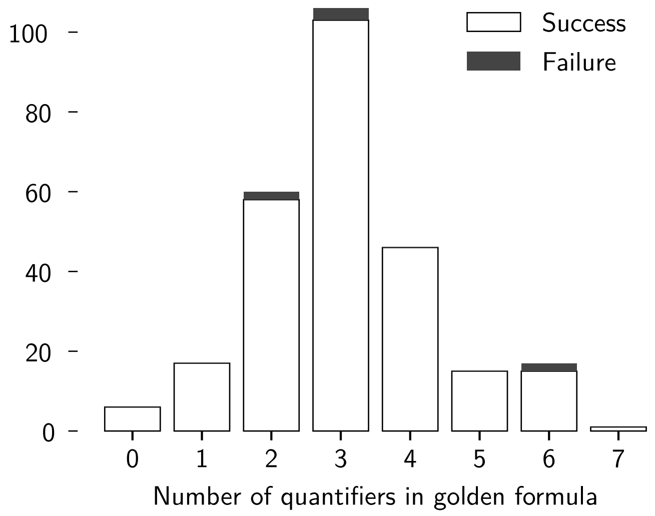

Figure 8.1: Stacked histogram of learning success (light) and failure (dark) by number of quantifiers in the golden formula. There were only failures in 2, 3, and 6 quantifier formulas.

8.2.1 Results and Discussion

We ran the learning process with function symbol depth bound $b = 1$ and overall timeout of 1 hour for each formula, including the time to explore prefixes, construct matrices, and check for equivalence using SMT solvers. Because the constraints only grow, each prefix is eliminated at most once, but multiple formulas of the same prefix are often generated with various matrices before the correct formula is found or the prefix is eliminated. The overall success rate of this process is 97.4%, and success rate by number of quantifiers in the golden formula can be seen in Figure 8.1.[2]

We give a *cactus plot* of the time to learn formulas in Figure 8.2. In this chart, examples are ordered by their time to learn along the x-axis, while the logarithmic y-axis shows the time to learn that formula. The shape of the chart depends on both the distribution of difficulty in the problems and the performance of the algorithm, so only general trends can be observed. This chart shows the time to solve is generally

[2]Our results are different from those reported in [28] due to several factors, including a different benchmark, improved SMT robustness, and a more efficient separation implementation.

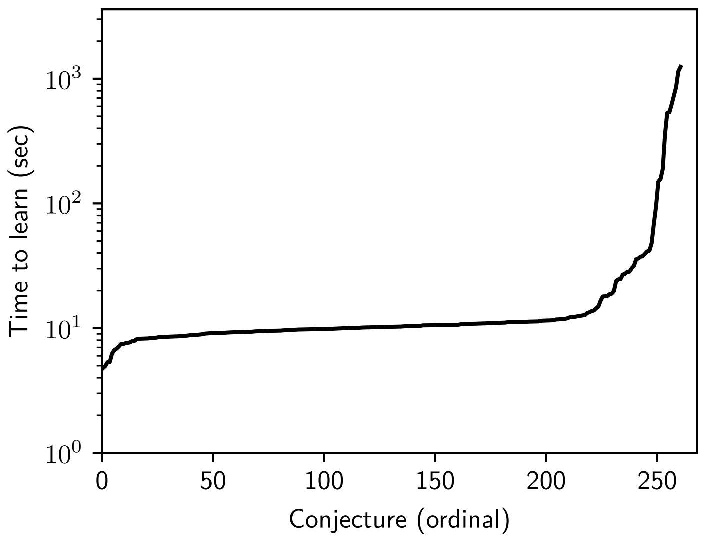

Figure 8.2: Cactus plot of time to learn formulas. Formulas are ordered by time to learn, and timeouts (>3600 sec) are the blank area on the right.

low for most formulas, with only a small number taking significantly longer than about 60 seconds.

Our results show that separation successfully learns a vast majority of the formulas; there were only 7 formulas that our technique did not learn. These failures are due to either SMT solver divergence or the separation algorithm getting lost in a large search space. The benchmark examples that had at least one failure are ring, ring-id-not-dead, fast-paxos-epr, and block-cache-async. Of these, our technique for invariant inference can solve all but block-cache-async (as presented below), which highlights the fact that there is not necessarily a single invariant for each problem. In the other direction, our technique does not find an overall invariant for some examples with all learnable conjuncts (e.g. vertical-paxos-epr). While this learning experiment gives us an indication of the performance of separation, it does not perfectly predict the results of invariant inference: inference can be harder due to e.g. implication constraints, or easier due to inferring a simpler invariant.

8.3 Evaluation of Invariant Inference

We evaluate our algorithm and other approaches on our benchmark of invariant inference problems. We discuss our experimental setup, and then the results of all the techniques. Finally, we present our ablation study.

8.3.1 Experimental Setup

We compare our algorithm to the techniques Swiss [19], IC3PO [16, 17], and PDR$^\forall$ [27]. We performed our experiments on a 56-thread machine with 64 GiB of RAM, with each experiment restricted to 16 hardware threads, 20GiB of RAM, and a 6 hour time limit. Specifically, we used systems with dual-socket Intel(R) Xeon(R) E5-2697 v3 CPUs running at 2.60GHz. To account for noise caused by randomness in seed selection and non-determinism from parallelism, we ran each algorithm 5 times and report the number of successes and the median time. PDR$^\forall$ and IC3PO are not designed to use parallelism, while Swiss and our technique make use of parallelism. For IC3PO, we use the better result from the two implementations [16] and [17]. For our technique, we ran the tool in universal-only, EPR, or full FOL mode as appropriate. For k-pDNF, we use $k = 1$ for universal prefixes and $k = 3$ otherwise. Our implementation uses five prefix categories (universal-only mode uses only the first two): (i) universal formulas, (ii) universal formulas with each sort appearing in at most two quantifiers, (iii) at most one quantifier alternation and each sort appearing in at most two quantifiers, (v) at most two quantifier alternations and each sort appearing in at most two quantifiers, and (vi) at most two quantifier alternations.

8.3.2 Results and Discussion

We present the results of our experiments in Table 8.2. In general, for examples that converge with both prior approaches and our technique, we typically match or exceed existing results in terms of time to solve each problem. Along with other techniques, we solve paxos-epr and flexible-paxos-epr, which are the simplest variants of Paxos in our benchmark, but nonetheless these examples represent a significant

Table 8.2: Experimental results, giving both the median wall-clock time (seconds) of run time and the number of trials successful, out of five for each technique. If there were less than 3 successful trials, we report the slowest successful trial, indicated by (>). A dash (-) indicates all trials failed or timed out after 6 hours (21600 seconds). A blank indicates no data.

Example	EPR	**Our**		Swiss		IC3PO		PDR$^\forall$	
lockserv	∀	19	5	9573	4	5	5	6	5
toy-consensus-forall	∀	4	5	22	5	4	5	4	5
ring-id	∀	7	5	192	5	81	5	20	5
sharded-kv	∀	8	5	17291	5	4	5	6	5
ticket	∀	23	5	-	0	-	0	22	5
learning-switch	∀	76	5	1744	4	29	5	94	5
consensus-wo-decide	∀	50	5	52	5	6	5	29	5
consensus-forall	∀	1908	5	80	5	15	5	104	5
cache	∀	2492	4	-	0	3906	5	2628	5
paxos-forall	∀	885	5	-	0	-	0	555	5
flexible-paxos-forall	∀	1961	5	-	0	1654	5	423	5
stoppable-paxos-forall	∀	7779	5	-	0	-	0	-	0
fast-paxos-forall	∀	-	0	-	0	-	0	20176	3
vertical-paxos-forall	∀	-	0	-	0	-	0	-	0
firewall	–	4	5	-	0	3	5		
sharded-kv-no-lost-keys	✓	4	5	9	5	4	5		
toy-consensus-epr	✓	4	5	10	5	4	5		
ring-id-not-dead	–	19	5	-	0	-	0		
consensus-epr	✓	37	5	57	5	28	5		
client-server-ae	✓	4	5	11	5	4	5		
client-server-db-ae	–	16	5	46	5	37	5		
hybrid-reliable-broadcast	–	178	5	-	0	-	0		
paxos-epr	✓	920	5	14332	4	-	0		
flexible-paxos-epr	✓	418	5	4928	5	-	0		
multi-paxos-epr	✓	4272	4	-	0	-	0		
stoppable-paxos-epr	✓	>18297	2	-	0	-	0		
fast-paxos-epr	✓	9630	3	-	0	-	0		
vertical-paxos-epr	✓	-	0	-	0	-	0		
block-cache-async	–	-	0	-	0	-	0		
bosco-3t-safety	✓	>11019[1]	1	-	0	-	0		

[1]With EPR push down enabled.

jump in complexity over the examples solved by the prior generation of PDR/IC3 techniques. Paxos and its variants are notable for having invariants with two quantifier alternations ($\forall\exists\forall$) and a maximum of 6 or 7 quantifiers. We uniquely solve multi-, fast-, and stoppable-paxos-epr, which add significant complexity in the number of sorts, symbols, and quantifiers required. Due to variations in seeds and the non-determinism of parallelism, our technique was only successful in some trials, but these results nevertheless demonstrate that our technique is capable of solving these examples. Our algorithm is unable to solve vertical-paxos-epr, as this example requires a 7 quantifier formula that, while solved in the learning experiment, proved too expensive for our IG solver.

For universal-only examples, our algorithm is able to solve all but one of the examples[3] solved by other techniques, and is able to solve one that others cannot. In some cases (e.g., consensus-forall), our solution is slower than other approaches, but on the whole our algorithm is competitive in a domain it is not specialized for.

8.3.3 Ablation Study

Table 8.3 presents an ablation study investigating the effect of various features of our technique. The first column of Table 8.3 repeats the full algorithm results, and the remaining columns report the performance with various features disabled one at a time. The most important individual contributions come from k-pDNF matrices and EPR. Using a 5-clause CNF instead of pDNF matrix (No pDNF) causes many difficult examples to fail and some (e.g., flexible-paxos-epr) to take significantly longer even when they do succeed.[4] This difference is due to k-pDNF matching the forms of the desired invariants more closely, while requiring less work from the separation algorithm (i.e., smaller k). Similarly, using full FOL mode instead of EPR (No EPR) leads to timeouts for all but the simplest Paxos variants, because the resulting lemmas are outside a decidable logic fragment and cause the SMT solvers to diverge. Note that it only takes a single solver divergence inside e.g. the pushing process to halt progress,

[3]fast-paxos-forall, which is solved by our technique in the ablation study, albeit rarely.

[4]With a single clause, there is no difference between CNF and k-pDNF so results are only given for existential problems.

Table 8.3: Ablation study, showing the results of our technique from Table 8.2 compared with disabling k-pDNF (No pDNF), EPR restrictions (No EPR), incremental SMT solving (No Inc.), and multi-block generalization (No Gen.). Entries are interpreted as in Table 8.2. Blanks indicate that the particular combination of example and experiment is not meaningful.

Example	**Our**	No pDNF	No EPR	No Inc.	No Gen.
lockserv	19 5			34 5	13 5
toy-consensus-forall	4 5			5 5	4 5
ring-id	7 5			11 5	13 5
sharded-kv	8 5			11 5	7 5
ticket	23 5			42 5	21 5
learning-switch	76 5			338 5	288 5
consensus-wo-decide	50 5			50 5	51 5
consensus-forall	1908 5			2154 5	558 5
cache	2492 4			>16826 2	13116 5
paxos-forall	885 5			1071 5	10488 4
flexible-paxos-forall	1961 5			1014 5	>4168 2
stoppable-paxos-forall	7779 5			2820 5	>18727 1
fast-paxos-forall	- 0			>16573 1	- 0
vertical-paxos-forall	- 0			- 0	- 0
firewall	4 5	4 5		4 5	4 5
sharded-kv-no-lost-keys	4 5	4 5	4 5	5 5	4 5
toy-consensus-epr	4 5	5 5	5 5	5 5	5 5
ring-id-not-dead	19 5	37 5		44 5	52 5
consensus-epr	37 5	126 5	724 5	45 5	233 5
client-server-ae	4 5	3 5	4 5	4 5	4 5
client-server-db-ae	16 5	13 5		20 5	10 5
hybrid-reliable-broadcast	178 5	98 5		173 5	629 5
paxos-epr	920 5	10135 4	>2895 1	609 5	3201 5
flexible-paxos-epr	418 5	13742 3	- 0	775 5	799 5
multi-paxos-epr	4272 4	>15176 1	- 0	15854 3	7326 4
stoppable-paxos-epr	>18297 2	- 0	- 0	>20659 1	>11946 1
fast-paxos-epr	9630 3	- 0	- 0	8976 3	>20871 2
vertical-paxos-epr	- 0	- 0	- 0	- 0	- 0
block-cache-async	- 0	- 0		- 0	>20038 2
bosco-3t-safety	>11019 1	- 0	- 0	>8581 1	>16689 1

Table 8.4: Parallel vs sequential comparison. Each of 5 trials ran with 3 or 48 hour timeouts, respectively. The number of successes, and the average number of IG queries in each trial (including failed ones) are given.

	Successes		IG Queries	
Example	Par.	Seq.	Par.	Seq.
paxos-epr	5	5	61	76
flexible-paxos-epr	5	5	64	72
multi-paxos-epr	3	1	67	84

as without a model we cannot drive the next step of inference. Incremental SMT queries (No Inc.) make the more difficult Paxos variants, and the universal-only cache example, succeed much more reliably. Incremental SMT only has a noticeable effect on the hardest queries, but it can make a subset of them much faster than waiting for the monolithic query from the solver. Multi-block generalization (No Gen.) makes many problems faster or more reliable, but disabling it does allow block-cache-async to succeed. Multi-block generalization can improve results by learning a more general lemma sooner, when there are multiple lemmas that could be added to the frame. When the desired lemmas are numerous but relatively simple (as in block-cache-async), the standard IG algorithm will find the right lemma, and the expense of multi-block generalization is not warranted.

To isolate the benefits of parallelism, we ran several examples in both parallel and serial mode with a proportionally larger timeout (Table 8.4). In both modes we use a single prefix category containing all prefixes, with the same static order over prefixes.[5] Beyond the wall-clock speedup, the parallel IG algorithm affects the quality of the learned lemmas, that is, how well they generalize and avoid overfitting. The computed lemma may be different in the parallel algorithm because the parallel algorithm effectively selects the first prefix from the static order to finish with a solution, not the first that is started. When there is more than one prefix that can solve the query, and those prefixes take significantly different amounts of time to solve, the parallel algorithm will be biased towards the solution that is solved faster. To estimate the quality of generalization from this effect, we count the total number of IG queries

[5]To make the comparison cleaner, we also disabled multi-block generalization.

performed by each trial and report the average over the five trials. In all examples, the parallel algorithm learns fewer lemmas overall, which suggests it generalizes better. We attribute this improved generalization to the implicit bias towards lemmas that are faster to discover. For the more complicated example (multi-paxos-epr), this difference has an impact on the success rate.

Chapter 9

Conclusion

We discuss the implications of our results and the possibilities for future work. Finally, we summarize our contributions.

9.1 Discussion

Separation is in a sense *dual* to SMT: separation takes in structures and produces a formula (or UNSEP), and SMT solvers take in formulas and produce a model (or UNSAT). When combined in a refinement loop, these two algorithms work together to produce invariants (ICE learning) or lemmas of invariants (PDR/IC3). Because separation is indifferent to the quantifier structure, we can use separation to produce formulas with quantifier alternations, which permits inference of invariants for complex protocols without needing these protocols to be rewritten to eliminate existentials. Separation, as introduced in [28], was the first technique able to infer invariants with quantifier alternations; subsequently, additional techniques have been developed that support alternations ([19, 16]).

Complex protocols require invariants that are quantified, and usually require alternations. While existentials can often be eliminated by hand, this requires transforming the system based on the invariant and is incompatible with the desire to increase automation and reduce the cost of verification. Therefore, our contributions are a significant step towards making verification practical for complex protocols and other software systems. Conventional wisdom says quantifiers are too expensive to be used in verification tasks, but our experimental results show that even the long-standing problem of invariant inference can be solved in the quantified domain. We infer invariants for systems that are not trivial; these systems are complex enough that it can take hours or days for a human to author a correct invariant. We hope our work sparks interest in quantified reasoning in verification, including both from the perspective of SMT solvers, and in how systems are designed, specified, and modeled.

9.2 Future Work

Practically, our invariant inference technique still has several gaps that need to be addressed before it can be readily integrated into verification practice: lack of support for theories, and use of EPR for reliable solutions. Solving these issues will greatly expand the potential applications for separation, and by extension the scope of automated verification techniques.

Separation is currently limited to uninterpreted first-order logic. While the restriction to first-order logic presents few impediments to modeling, most interesting systems (programs, protocols, hardware models, etc.) require interpreted theories, such as bitvectors, theory of arrays, arithmetic, sequences, strings, etc. Such theories are useful for modeling both the data that algorithms process and crucially the representation of data structures. While the abstract definition of separation does not change when theories are added, this addition will require a new algorithm for separation. In particular, our algorithm makes the assumption of a finite signature, which is violated by e.g., arithmetic where each number is essentially a distinct constant. Further, models from an SMT solver are no longer guaranteed to be small or even finite, so the algorithm cannot in general compute all assignments of model elements to quantified variables. Even in domains that are not infinite, such as bitvectors, the number of elements can be impractically large (e.g., 2^n elements for n-bit bitvectors). Future extensions of separation to interpreted domains will need some way to focus on the relevant elements from such large domains.

The large or infinite model issue is related to the use of EPR. Even in uninterpreted domains, formulas such as $\forall x{:}S.\, \exists y{:}S.\, \varphi$ create Skolem functions in an SMT solver that can generate an infinite number of terms. The use of EPR is awkward for invariant inference, as the transition system must sometimes be transformed to admit an EPR invariant, using knowledge of the final invariant. Solving this issue may involve generating more information from the separation algorithm, such as a finite set of instantiations to use for the quantifiers in the separator. Fundamentally, using EPR gives us decidability at the expense of expressiveness, and an ideal solution to this problem will restore expressiveness while, if not preserving decidability, increasing the reliability of SMT solving in domains of interest. It is likely that a solution to the EPR problem will also play a role in supporting theories, as similar issues of model size arise.

9.3 Summary

We present first-order quantified separation, a new computational primitive for generating formulas from examples. Separation naturally produces quantified formulas with quantifier alternations. We present a practical algorithm for solving separation using a SAT solver, and describe several optimizations. We show that separation is NP-complete. We use separation to build a state-of-the-art invariant inference algorithm based on PDR/IC3. Our algorithm is able to find invariants for complex distributed protocols that are unsolved by other techniques. Our work is a significant step towards practical automation using quantified formulas in software verification.

References

[1] Aws Albarghouthi and Kenneth L. McMillan. Beautiful interpolants. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 313–329, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[2] 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. Snowbird, Utah.

[3] Idan Berkovits, Marijana Lazic, Giuliano Losa, Oded Padon, and Sharon Shoham. Verification of threshold-based distributed algorithms by decomposition to decidable logics. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, pages 245–266, 2019.

[4] Aaron R. Bradley. Sat-based model checking without unrolling. In Ranjit Jhala and David Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 70–87, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[5] Nader H. Bshouty. Exact learning boolean functions via the monotone theory. *Inf. Comput.*, 123(1):146–153, November 1995.

[6] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[7] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] Samuel Drews and Aws Albarghouthi. Effectively propositional interpolants. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 210–229. Springer, 2016.

[9] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient implementation of property directed reachability. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 125–134, 2011.

[10] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. Quantified invariants via syntax-guided synthesis. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 259–277. Springer, 2019.

[11] Yotam M. Y. Feldman, Neil Immerman, Mooly Sagiv, and Sharon Shoham. Complexity and information in invariant inference. *PACMPL*, 4(POPL):5:1–5:29, 2020.

[12] Yotam M. Y. Feldman, Oded Padon, Neil Immerman, Mooly Sagiv, and Sharon Shoham. Bounded quantifier instantiation for checking inductive invariants. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 76–95, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[13] Yotam M. Y. Feldman, James R. Wilcox, Sharon Shoham, and Mooly Sagiv. Inferring inductive invariants from phase structures. In Isil Dillig and Serdar Tasiran,

editors, *Computer Aided Verification*, pages 405–425, Cham, 2019. Springer International Publishing.

[14] GW Ford and GE Uhlenbeck. Combinatorial problems in the theory of graphs. In *Proc Natl Acad Sci USA*, pages 163–167, 1957.

[15] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. Ice: a robust framework for learning invariants. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 69–87, Cham, 2014. Springer International Publishing.

[16] Aman Goel and Karem A. Sakallah. On Symmetry and Quantification: A New Approach to Verify Distributed Protocols. In *13th Annual NASA Formal Methods Symposium (NFM 2021)*, Langley, Virginia, May 2021. (*Accepted*).

[17] Aman Goel and Karem A. Sakallah. Towards an Automatic Proof of Lamport's Paxos. In *FMCAD 2021*, 2021. (*Accepted*).

[18] Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. Quantifiers on demand. In Shuvendu K. Lahiri and Chao Wang, editors, *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, volume 11138 of *Lecture Notes in Computer Science*, pages 248–266. Springer, 2018.

[19] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. Finding invariants of distributed systems: It's a small (enough) world after all. In James Mickens and Renata Teixeira, editors, *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, pages 115–131. USENIX Association, 2021.

[20] Z. Hassan, A. R. Bradley, and F. Somenzi. Better generalization in IC3. In *2013 Formal Methods in Computer-Aided Design*, pages 157–164, 2013.

[21] Jaakko Hintikka. Game-theoretical semantics: insights and prospects. *Notre Dame J. Formal Logic*, 23(2):219–241, 04 1982.

[22] Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2012.

[23] Neil Immerman. *Descriptive Complexity.* Graduate Texts in Computer Science. Springer, 1999.

[24] Neil Immerman and Eric Lander. Describing graphs: A first-order approach to graph canonization. In Alan Selman, editor, *Complexity Theory Retrospective*, pages 59–81. Springer-Verlag, 1990.

[25] Alexander Ivrii and Arie Gurfinkel. Pushing to the top. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, FMCAD '15, page 65–72, Austin, Texas, 2015. FMCAD Inc.

[26] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2012.

[27] 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):7:1–7:33, March 2017.

[28] 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.

[29] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. *Formal Methods Syst. Des.*, 48(3):175–205, 2016.

[30] Hari Govind Vediramana Krishnan, Yuting Chen, Sharon Shoham, and Arie Gurfinkel. Global guidance for local generalization in model checking. In *Computer*

Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II, pages 101–125, 2020.

[31] K. Rustan M. Leino. Developing verified programs with dafny. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1488–1490, Piscataway, NJ, USA, 2013. IEEE Press.

[32] 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 Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 370–384. ACM, 2019.

[33] Matteo Marescotti, Arie Gurfinkel, Antti Eero Johannes Hyvärinen, and Natasha Sharygina. Designing parallel PDR. In Daryl Stewart and Georg Weissenbacher, editors, *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 156–163. IEEE, 2017.

[34] Kenneth L. McMillan. Interpolation and sat-based model checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[35] Kenneth L. McMillan. Lazy abstraction with interpolants. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.

[36] mypyvy repository. `https://github.com/wilcoxjay/mypyvy`.

[37] Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. Reducing liveness to safety in first-order logic. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[38] 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):1–31, Oct 2017.

[39] 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*, PLDI '16, pages 614–630, New York, NY, USA, 2016. ACM.

[40] Rahul Sharma and Alex Aiken. From invariant checking to invariant inference using randomized search. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, pages 88–105, Berlin, Heidelberg, 2014. Springer-Verlag.

[41] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V. Nori. Verification as learning geometric concepts. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis*, pages 388–411, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[42] Rahul Sharma, Aditya V. Nori, and Alex Aiken. Bias-variance tradeoffs in program analysis. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 127–137, New York, NY, USA, 2014. ACM.

[43] 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*, PLDI 2018, page 662–677, New York, NY, USA, 2018. Association for Computing Machinery.

[44] V. Vapnik and A. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability & Its Applications*, 16(2):264–280, 1971.

[45] Hongce Zhang, Aarti Gupta, and Sharad Malik. Syntax-guided synthesis for lemma generation in hardware model checking. In Fritz Henglein, Sharon Shoham, and Yakir Vizel, editors, *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings*, volume 12597 of *Lecture Notes in Computer Science*, pages 325–349. Springer, 2021.