Finding Invariants of Distributed Systems: It's a Small (Enough) World After All

Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno
Carnegie Mellon University

Abstract

Today's distributed systems are increasingly complex, leading to subtle bugs that are difficult to detect with standard testing methods. Formal verification can provably rule out such bugs, but historically it has been excessively labor intensive. For distributed systems, recent work shows that, given a correct inductive invariant, nearly all other proof work can be automated; however, the *construction* of such invariants is still a difficult manual task.

In this paper, we demonstrate a new methodology for automating the construction of inductive invariants, given as input a (formal) description of the distributed system and a desired safety condition. Our system performs an exhaustive search within a given space of candidate invariants in order to find and verify inductive invariants which suffice to prove the safety condition. Central to our ability to search efficiently is our algorithm's ability to learn from counterexamples whenever a candidate fails to be invariant, allowing us to check the remaining candidates more efficiently. We hypothesize that many distributed systems, even complex ones, may have concise invariants that make this approach practical, and in support of this, we show that our system is able to identify and verify inductive invariants for the Paxos protocol, which proved too complex for previous work.

1 Introduction

The world increasingly relies on distributed computer systems, but the correctness and reliability of these systems depend on the imperfect coverage of testing. To obtain strong guarantees, developers are starting to turn to formal verification techniques, both in research [3, 21, 24, 31, 46, 54] and industry [38, 58]. These techniques can, in theory, prove that all possible execution traces of the system conform to a high-level safety specification (e.g., all nodes agree on the next input value, or no two nodes hold a lock at the same time).

However, verifying the safety of distributed systems can be extremely labor-intensive, especially when using general-purpose theorem provers [21, 45, 54]. For example, Hawblitzel et al. [21] report that the effort to build and verify two distributed systems (including their protocols and implementations) required 3.7 person-years, and their safety proofs (over 19K lines of code) account for ∼40% of the total codebase. To reduce this cost, some work has developed specialized languages that either restrict the kinds of systems that can be encoded [12, 13, 36, 53] (e.g., the protocol must proceed in synchronous rounds), or restrict the language used to describe the systems' properties [42]. In exchange for these restrictions, much of the safety proof can be dispatched automatically.

In all of these systems, the core of the safety proof requires identifying system *invariants*, and even with tools that can automate all other proof work, finding these invariants still relies on human labor and ingenuity. Anecdotally, this is a challenging task even for researchers [34], requiring days for toy systems and months for complex protocols like Paxos [29].

Ideally, we would automate invariant discovery, but theoretical results show that even in languages where *checking* an invariant is decidable [42], *finding* such an invariant is not decidable [40], meaning that no algorithm can guarantee that it will find an invariant for arbitrary protocols, even if they are indeed safe. Hence, we must turn to domain-specific insights to develop a methodology which can apply to the specific kinds of distributed systems developed in practice.

Recent approaches have been proposed to automatically infer invariants for distributed systems [27, 34]. For instance, the I4 [34] system observes that the invariants for some distributed systems are scale-invariant. Hence they ask the developer to specify appropriate finite-model parameters and to concretize some of the protocol variables. I4 then invokes a specialized model checker [19] that can deduce invariants on finite, fixed-size instances. It then employs various heuristics to generalize those invariants to arbitrary instances. Unfortunately, I4 cannot discover invariants that use existential quantifiers. Furthermore, since the model checker is a black box, the developer has no recourse when it fails. More recently, Koenig et al. [27] extend the IC3/PDR algorithm [6, 14] to infer invariants with existential quantifiers. However, neither they nor I4 can handle complex protocols like Paxos [29], for which the simplest known safety proofs require many invariants, including some with existential quantifiers.

To tackle complex protocols like Paxos, we explore an alternate approach based on our "small world" hypothesis: in many practical systems, the invariants should be relatively concise. After all, these protocols are designed by humans who have some (finite) intuition for why the protocol is correct. They are clearly not beyond human comprehension. If accurate, this

hypothesis suggests we are faced with a finite space of possible invariants, which we can potentially search exhaustively.

Of course, finite is not the same as small. For a protocol like Paxos, there could be over 100 billion candidate invariants with just six terms. Even if it only took a millisecond to test each invariant, a brute force search of the entire space would require over three computation-years.

Hence, this paper presents the Small World Invariant Search System (SWISS), our system for automatically proving the safety of distributed systems by efficiently searching through the space of succinct invariants. At its core, SWISS learns from counterexample models obtained from failed invariant candidates to speed up future invariant checks. SWISS uses an SMT solver to perform these checks. SWISS also exploits various symmetries and parallelism to reduce search time. Further, for a class of invariants (described formally in §3.4) SWISS is guaranteed to find an invariant when one exists. SWISS also supports user-supplied guidance to make the search more efficient. Moreover, unlike prior approaches, even when SWISS does not produce a safety proof within a given time limit, the user still benefits from partial invariants that SWISS did find, as they can use those invariants as a starting point for finding complete invariants. The net effect is that SWISS is the *first approach that can automatically prove the safety of Paxos*, and it does so in around 4 hours on an 8-core machine. If the user provides some light guidance (§5.3.3), then the time decreases to 20 minutes.

We compare SWISS to I4 [34] and Koenig et al. [27] on a large variety of protocols (§5). None of the approaches fully dominates the others, neither in protocols solved nor in runtime. For instance, for protocols that only require invariants without existential quantifiers, I4 is generally fastest. However, for some complex distributed protocols with many invariants that contain existential quantifiers, SWISS outperforms the other tools, and in particular, it is the first to automatically prove the safety of Paxos and two variants of Paxos: Multi-Paxos and Flexible-Paxos [22]. There are still some variants of Paxos that SWISS cannot fully prove in a reasonable amount of time. However, even in these cases, SWISS still finds many partial invariants that may help the user complete the proof.

In summary, our key contributions are as follows.

- We propose and evaluate the *Small World Hypothesis*: that many distributed systems we care for can be proven safe using a sequence of concisely specified invariants.
- We present SWISS, a methodology for automatically proving the safety of distributed systems by efficiently searching through the space of candidate invariants, guided by knowledge learned from counterexamples encountered during the search. We report both our successful *and* unsuccessful optimizations.
- To our knowledge, SWISS is the first approach to identify and verify inductive invariants that prove the safety of Paxos in an *automated* fashion.

2 Background

We briefly present background on formalizing and verifying distributed systems.

2.1 Proving Safety Conditions Via Inductive Invariants

We aim to prove *safety conditions*. A safety condition is a desired property that ought to hold true at any point in a system's execution. For an exclusion lock, for example, the safety condition might be that no two agents hold a lock at the same time. For a consensus protocol such as Paxos, the condition would be that no two machines decide on different results. These safety conditions are in contrast with *liveness conditions*, which say that the system eventually performs a useful action. We do not currently consider liveness conditions.

To formalize a distributed system, one must formally describe the internal state of the participating nodes, the state of the network (packets in-flight between nodes), the initial conditions of the system, and the ways the system can evolve. Following standard practice, we formalize the latter as a sequence of atomic actions that update the system state [28].

Given this formal description, our goal then becomes to prove that the safety condition will hold for any reachable state in any possible execution of the system. The typical strategy for such a proof is to find an *inductive invariant*. An inductive invariant should **(i)** hold on all possible initial states of the system and **(ii)** continue to hold when the system transitions to a new state from a state where the invariant held.

If we could prove that the safety condition was inductive, we would be done; unfortunately, this is rarely the case for non-trivial systems. Instead, we typically must find an invariant which is stronger than the safety condition and then show this invariant is both inductive *and* implies the safety condition.

2.2 A Running Example: Simple Decentralized Lock

To make the verification process more concrete, we present a toy example: a Simple Decentralized Lock (SDL) protocol. SDL supports a single mutual-exclusion lock that is shared among multiple nodes. A node with the lock can send the lock to another machine via a message over the network, which, in this simple example only, does not permit packet duplication.

Figure 1 formally describes the SDL protocol. The state at a snapshot in time is represented by two relations: message and lock. The value message(src, dst) indicates a message in the network from machine *src* to machine *dst*, while lock$(node)$ indicates that *node* believes it holds the lock.

In the protocol's initial state (the **init** lines), the network is empty, and only one node ($start\_node$) holds the lock.

The transitions are written in RML [42], an abstract language for describing system transitions. SDL has two transition actions: `send` and `recv`. In `send`, a node *with the lock* can atomically release the lock and send a packet to another machine. In `recv`, a node can receive a packet (removing it from the network) and accept the lock.

The safety condition (Figure 2) we wish to verify for SDL is

type node

relation message(src : node, dst : node): bool
relation lock(N : node): bool

init $\forall src, dst.\ \neg message(src, dst)$
init $\exists startnode$: node.
  $has\_lock(startnode)$
  $\wedge\ \forall N$: node. $(N \neq startnode \implies \neg \text{lock}(N))$

action `send`(src: node, dst: node) {
  require lock(src);
  message(src, dst) := $true$;
  lock(src) := $false$;
}

action `recv`(src: node, dst: node) {
  require message(src, dst);
  message(src, dst) := $false$;
  lock(dst) := $true$;
}

Figure 1: **The Simple Decentralized Lock Protocol**

SDL safety condition

$$\forall n_1, n_2 : \text{node}.\ \text{lock}(n_1) \wedge \text{lock}(n_2) \implies n_1 = n_2$$

SDL inductive invariant

$$(\forall n_1, n_2 : \text{node}.\ \text{lock}(n_1) \wedge \text{lock}(n_2) \implies n_1 = n_2) \wedge$$
$$(\forall n_1, n_2, n_3 : \text{node}.\ \neg(\text{lock}(n_1) \wedge \text{message}(n_2, n_3))) \wedge$$
$$(\forall n_1, n_2, n_3, n_4 : \text{node}.\ \text{message}(n_1, n_2) \wedge \text{message}(n_3, n_4)$$
$$\implies n_1 = n_3 \wedge n_2 = n_4)$$

Figure 2: **Safety Condition and Invariants for the SDL**

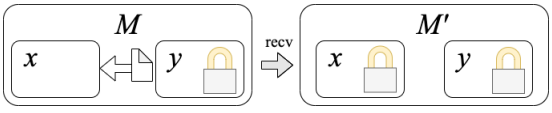

Figure 3: Two models which demonstrate that the safety condition (Figure 2) for SDL is not inductive on its own. M and M' each have a *domain* of two nodes, n_1 and n_2. In M, n_2 holds the lock, while a packet is in-flight from n_2 to n_1. M can transition to M' via the `recv` action. M does not violate the safety condition, but M' does (n_1 and n_2 both hold the lock).

that no two nodes ever hold the lock at the same time. Figure 3 shows that this condition is *not* inductive. There is a state, M, which satisfies the safety condition (only one machine holds the lock) which can transition to a state, M' (via a `recv` on y) which does not satisfy the safety condition (since two machines hold the lock). Figure 2 shows a stronger predicate which also rules out the first state. This predicate is both inductive and (trivially) implies the safety condition; thus, it completes our safety proof. It states that **(i)** no two machines hold the lock, **(ii)** no machine holds the lock while a message exists, and **(iii)** no two messages exist at the same time.

2.3 Formal Notation

To keep our subsequent discussion precise we introduce some additional notation. Formally, a *transition system* is a triple $\mathcal{T} = (\Sigma, INIT, TR)$, where Σ defines the types and relations representing the state of a system, and $INIT$ and TR are predicates describing the initial state of the system and allowed transitions of the system, respectively. In our SDL example, Σ contains the type `node` and the relations lock and message.

Since TR relates two states, we will use primes to indicate the new state, e.g., $TR := (x' = x + 1)$ represents a transition that increments x by one.

A *model* M represents a single possible state of the system. It contains an assignment for each variable and each possible evaluation of the system's relations. For a predicate P, we write $M \models P$ if the predicate P evaluates to $true$ on model M. When P is a predicate over two states, we write $(M, M') \models P$, e.g., we write $(M, M') \models TR$ if M can transition to M'.

We say that M is *reachable* if there exists a sequence $M_0, \ldots, M_k$ where $M_0 \models INIT$, $M_k = M$, and $(M_i, M_{i+1}) \models TR$ for all i. A formula S is said to be *safe* if for all reachable M, we have $M \models S$.

We say that I (a formula over Σ) is *inductive* if we can prove that $INIT \implies I$ and $TR \wedge I \implies I'$ (i.e., if I is true and the system can take a transition to a new state, then I holds there as well). Here, we use I' to denote the predicate I evaluated on the second state. It is clear that any inductive invariant I will be safe. Therefore, proving that S is safe amounts to finding an inductive invariant I such that $I \implies S$; equivalently, to find a formula I such that $I \wedge S$ is an inductive invariant.

2.4 Decidability of Inductiveness

Given a candidate invariant I, we must prove that **(i)** $INIT \implies I$, **(ii)** $TR \wedge I \implies I'$, and **(iii)** $I \implies S$. We call these *verification conditions*. To check that a verification condition P holds for all possible models, we can show that $\neg P$ is unsatisfiable; i.e., there is no model M such that $M \models \neg P$.

Checking the validity of arbitrary first-order logic formulas is undecidable [52], but prior work [41, 42] shows that many distributed systems, including multiple variants of Paxos, can be encoded in RML [42], which can be translated to a restricted class of formulas where satisfiability *is* decidable. In particular, the class of *effectively propositional* (EPR) formulas, also known as the BernaysSchönfinkel class, are the class of formulas that can be written in a form with quantifier prefix $\exists^*\forall^*$ and no function symbols. Satisfiability for this class of formulas is decidable [33, 43].

By ensuring our verification conditions lie within this class, we can always either verify the predicate $INIT \implies I$ or find a satisfying instance for $INIT \wedge \neg I$. Likewise, we can either verify the predicate $TR \wedge I \implies I'$ or find a pair (M, M') where $(M, M') \models TR \wedge I \wedge \neg I'$ (e.g., the pair in Figure 3). In either

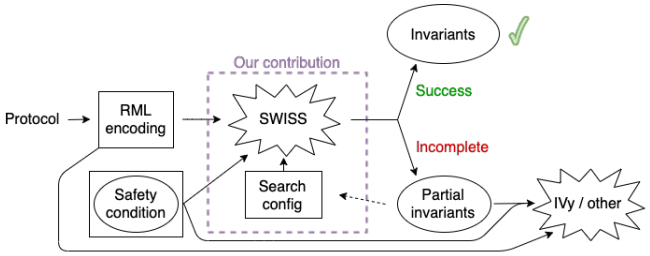

Figure 4: **SWISS Overview**. The intended workflow for using SWISS to synthesize an invariant and safety proof. A rectangle indicates a machine-readable, human-supplied input. An oval represents a collection of first-order predicates.

case, we obtain a concrete counterexample.

EPR, as stated, is a bit too restrictive; it allows only relations (not general functions), and it does not allow for invariants with quantifier alternation. However, EPR can be extended to include *stratified* function symbols—functions for which the edges from input types to output types form no cycles—while maintaining its decidability. In the same way, we can allow stratified alternation of universal and existential quantifiers in a fragment called *the extended EPR fragment* [41]. To keep our verification conditions within this class, we must impose some restrictions on the quantifier alternations which appear in I; these restrictions are determined by the shape of the protocol under consideration, in particular, the quantifier alternations which appear in $INIT$ and TR.

3 Overview: The SWISS Algorithm

SWISS is an algorithm for inferring inductive invariants of a protocol in order to prove a desired safety condition. The intended usage of SWISS is shown in Figure 4. First, the user provides an RML-encoding [42] of the protocol, a desired safety condition, and a specification of the search space (§4.1.1). SWISS either succeeds with an invariant that proves the safety condition, or it fails, with only partial invariants generated. In that case, the user may choose how to continue: they might try SWISS again with a different search configuration, or they might continue with other means, e.g., the interactive invariant-finding tool IVy [42], using the partial invariants as a starting point.

For example, suppose the user is interested in the SDL protocol (§2) and wants to prove the lock-exclusitivity safety condition. They would first encode the SDL protocol into a machine-readable RML specification (Figure 1). Protocols are often concise, although in some cases it is challenging to produce a specification where the inductive invariant will be in EPR [41]. However, we consider this out of scope for SWISS.

Next, the user would write the exclusitivity condition as a predicate (Figure 2). They would also choose the space of predicates to search over (§4.1.1). If the user knows nothing about the protocol, they would likely choose the most general option, to generate templates automatically. If SWISS succeeds, then they will know that the lock-exclusitivity safety condition is true, and they can use the invariants from SWISS's output to validate that SWISS ran correctly.

If SWISS does not succeed, there are a few possible paths. For example, it might be that SWISS does not complete in a reasonable amount of time, in which case the user might choose to restrict the search space. For example, they might have some idea of what the invariant should look like because they have worked with similar locking protocols previously. Alternatively, if SWISS completed quickly but did not succeed, the user might choose a broader search space.

Finally, they might choose to take the incomplete invariants generated by SWISS and attempt to complete them through other means. Even though SWISS did not succeed, the user could learn useful information about the protocol from these incomplete invariants.

3.1 High-Level Algorithm

We begin with a high-level overview of our algorithm for finding the inductive invariants needed to prove safety conditions. Section 4 then explains how we make the algorithm scale to large search spaces.

SWISS takes as input **(i)** a transition system $\mathcal{T}$ encoded via RML (§2.2) **(ii)** a safety condition S, and **(iii)** a configuration of the search space (§4.1.1). In this section, we refer to search spaces with the symbols $\mathcal{B}$ and $\mathcal{F}$, which here may be viewed as arbitrary sets of first-order predicates.

SWISS is designed to exploit our hypothesis that a distributed system designed by humans will have a concise invariant, or a larger invariant composed of concise invariants. After all, the designer presumably has a finite intuition for the correctness of their system, either as a whole or as the conjunction of correct subsystems or subproperties.

Internally, SWISS uses different algorithms to target these two possible invariant styles. One algorithm, Finisher (§3.2), tries to directly find one inductive invariant that proves the safety condition. Hence any invariant it finds will necessarily complete the safety proof. Using the safety condition as a target helps Finisher search the invariant space efficiently. However, for complex protocols, searching for the entire system invariant in one shot is infeasible; e.g., a human-derived invariant for Paxos has 10 conjuncts with 34 terms, corresponding to a search space of over 10^{75} candidate invariants.

Hence, SWISS employs a second algorithm, Breadth (§3.3), that greedily searches for as many protocol invariants as possible within a finite space $\mathcal{B}$, without requiring that they directly prove the safety condition. We run Breadth multiple times so that invariants may build on each other: once Breadth finds an invariant P, the next run can then find an invariant Q that is inductive relative to P, even when Q might not be inductive on its own. More formally, we say that Q is *relatively inductive* with respect to P if $INIT \implies Q$ and $TR \wedge P \wedge Q \implies Q'$.

Algorithm 1: Solve $(\mathcal{T}, S, \mathcal{B}, \mathcal{F})$

```
invariants ⟵ {};
while true do
    newInvariants ⟵ Breadth(T, invariants, B);
    if newInvariants imply safety then
        return newInvariants
    if newInvariants = invariants then
        break;
    invariants ⟵ newInvariants;
return Finisher(T, invariants, S, F) ∪ invariants
```

The Breadth loop ultimately builds an invariant of the form $P_1 \wedge \cdots \wedge P_n$, where each P_i is relatively inductive to $P_1 \wedge \cdots \wedge P_{i-1}$. In practice (§5), without the safety condition guiding it, Breadth is slower than Finisher, but it can incrementally construct a larger invariant than Finisher can.

To benefit from the strengths of both Breadth and Finisher, SWISS's top-level Solve (Algorithm 1) combines them. It takes as input a transition system, $\mathcal{T}$, and a desired safety condition S. It also takes in two spaces of candidate invariants, $\mathcal{B}$ and $\mathcal{F}$, for Breadth and Finisher, respectively, to search. In our implementation, these spaces are defined through a combination of the protocol description and (potentially) user input (§4.1). Solve runs Breadth over $\mathcal{B}$ until no new invariants are produced, and then it runs Finisher, if necessary, to find an additional invariant needed to complete the safety proof. Section 3.4 summarizes SWISS's coverage guarantee.

3.2 The Finisher Algorithm

Finisher aims to find a single invariant P which proves a safety condition S. More formally, it tries to solve:

Task 1 (Conjecture-proving task.) *Given a transition system $\mathcal{T}$, formulas $I_1, \ldots, I_n$, already established (or assumed) to be invariant, and a conjectured safety condition S, find an invariant predicate P such that $P \wedge S$ is inductive relative to $I_1 \wedge \cdots \wedge I_n$.*

Evaluating a candidate invariant P requires checking the validity of the following *verification conditions* (VCs):

$$\begin{aligned} INIT &\implies S \\ INIT &\implies P \\ TR \wedge I_1 \wedge \cdots \wedge I_n \wedge S \wedge P &\implies S' \\ TR \wedge I_1 \wedge \cdots \wedge I_n \wedge S \wedge P &\implies P' \end{aligned}$$

Since the $INIT \implies S$ condition does not depend on P, we can check it once in advance of evaluating any candidat invariant.

Since we consider protocols expressed in RML (§2.4), checking the validity of these VCs is decidable, and in practice, typically quite efficient with modern SMT solvers. Hence, for a candidate invariant P, we can run a subroutine CheckVCsF to determine either that the VCs above hold, or that a finite counterexample shows they do not. As Algorithm 2 shows, rather than simply check the VCs above for each candidate predicate P in $\mathcal{F}$, Finisher accumulates a collection of counterexamples from failed candidates. As we describe in §4.2, we use these counterexamples to filter subsequent candidates, as our counterexample check is orders of magnitude faster than the VC check.

Algorithm 2: Finisher$(\mathcal{T}, \{I_1, \ldots, I_n\}, S, \mathcal{F})$

```
cexamples ⟵ {};
for P ∈ F do
    if ∀cex ∈ cexamples . Passes(P, cex) then
        cex ⟵ CheckVCsF(T, {I_1, ..., I_n}, S, P);
        if cex is None then
            return P;
        else
            cexamples ⟵ cexamples ∪ {cex};
return None
```

Algorithm 3: Breadth$(\mathcal{T}, \{I_1, \ldots, I_n\}, S, \mathcal{B})$

```
cexamples ⟵ {};
allInv ⟵ {I_1, ..., I_n};
indInv ⟵ {I_1, ..., I_n};
for P ∈ B do
    if ∀I ∈ allInv . ¬FastImplies(I, P) then
        if ∀cex ∈ cexamples . Passes(P, cex) then
            cex ⟵ CheckVCsB(T, {I_1, ..., I_n}, S, P);
            if cex is None then
                allInv ⟵ allInv ∪ {P};
                if ¬ Redundant (P, indInv) then
                    indInv ⟵ indInv ∪ {P};
            else
                cexamples ⟵ cexamples ∪ {cex};
return indInv;
```

3.3 The Breadth Algorithm

In the Breadth algorithm, our goal is simply to find as many invariants as possible. More formally, we wish to solve the following task.

Task 2 (Invariant-finding task.) *Given a transition system $\mathcal{T}$ and formulas $I_1, \ldots, I_n$, already established (or assumed) to be invariant, find any invariant predicate P which is inductive relative to $I_1 \wedge \cdots \wedge I_n$.*

The corresponding VCs are as follows.

$$\begin{aligned} INIT &\implies P \\ TR \wedge I_1 \wedge \cdots \wedge I_n \wedge P &\implies P' \end{aligned}$$

Naively, we could start with an algorithm similar to Finisher, but use the VCs above instead of Finisher's. However, this would result in a highly inefficient search since as stated, the invariant-finding task permits many tautological solutions, such as $true$ or I_i. More generally, if P is any predicate such that $I_1 \wedge \cdots \wedge I_n \implies P$, then P will be invariant, but we do not actually learn anything about the protocol by finding P: P does not rule out any states which were not ruled out by the I_i. We call such a P a *redundant invariant* with respect to $I_1 \wedge \cdots \wedge I_n$. For example, $\forall x.f(x) \vee g(x)$ is redundant with respect to $\forall x.f(x)$. As the number of terms in our search space increases, the number of redundant invariants increases exponentially. Furthermore, since any redundant invariant is, in fact, inductive, it will always pass our counterexample filters, guaranteeing an expensive VC check for each.

We devised two ways to cope with redundant invariants. First, we maintain a set *indInv* of non-redundant invariants, and whenever we find a new invariant, we explicitly check whether it is redundant with *indInv*. Second, we also track *allInv*, i.e., all invariants we find—including the redundant ones—and use these to syntactically filter future candidates by performing quick checks for logical implication using a subroutine FastImplies(I,P) described in §4.4. These checks are vastly cheaper than SMT calls.

As described thus far, Breadth works without any knowledge of the safety condition S that we aim to prove about our system. However, we observe that we will eventually need S to be an invariant of the system, so we strengthen the second VC above to assume S is true in the initial state.

$$TR \wedge I_1 \wedge \cdots \wedge I_n \wedge S \wedge P \implies P'$$

Algorithm 3 brings all of this together. This algorithm finds exactly the invariants from the space $\mathcal{B}$ which are invariant with respect to the original input invariants. More precisely, if $P \in \mathcal{B}$ is invariant with respect to the inputs, then Breadth will output a set of predicates whose conjunction implies P.

3.4 SWISS Coverage

SWISS is guaranteed to prove a safety condition S as long as the system invariant conforms to the following form.

Claim 1 Solve$(\mathcal{T},S,\mathcal{B},\mathcal{F})$ *will always succeed at proving the conjectured safety condition S provided there exist invariants* $I_1,\cdots,I_n$ *such that:*

- $I_i \in \mathcal{B}$ *for* $1 \leq j \leq n-1$.
- $I_n \in \mathcal{F}$.
- $I_1 \wedge \ldots \wedge I_j$ *is inductive relative to S for* $1 \leq j \leq n-1$.
- $I_1 \wedge \ldots \wedge I_n \wedge S$ *is inductive.*

The claim follows from inspection of Algorithms 1-3 and the fact that our counterexample and implication filters will never eliminate a valid invariant.

4 Making Invariant Exploration Scale

While the algorithms described in §3 would theoretically suffice to find invariants, implemented naively they are impractically slow. Hence, we present crucial steps we take to reduce the search space (§4.1-§4.4) and optimize the overall process (§4.5). For the sake of completeness, we also present three optimizations that our evaluation has shown *do not* improve performance in this domain (Appendix A).

4.1 Exploiting User Guidance & Candidate Symmetries

The Finisher and Breadth algorithms each take as input a space of candidate invariants to explore (§4.1.1). These spaces often contain many invariants that are identical modulo symmetries, so symmetry pruning is critical (§4.1.2).

4.1.1 Defining Candidate Spaces

SWISS searches for invariants based on *invariant templates*. Intuitively, a template defines the rough shape of a class of invariants (e.g., the number and types of the quantified variables, and a bound on the formula's size).

By default, SWISS automatically defines invariant templates based on the protocol description and the safety condition, as well as user-supplied upper bounds on the formula size. SWISS then enumerates all template spaces within these constraints, and the search space $\mathcal{B}$ or $\mathcal{F}$ is defined as the union of these spaces. Finisher prioritizes these templates in order of increasing size. Thus, if a small invariant exists, Finisher will find it without having to search the entire space.

For many protocols, this fully automatic approach suffices to produce a safety proof. For protocols where this automated search runs slower than desired, the user can specify a particular template T, rather than have SWISS enumerate all templates. As §5.3.3 demonstrates, such user guidance can dramatically speed up the search process, even if the user guesses a few incorrect templates before the right one.

More formally, every candidate invariant P that we consider is a sequence of quantifiers followed by a quantifier-free expression E. The expression E is a tree of conjunctions and disjunctions of *terms* C, expressed over values V, which is either a name and a type (e.g., $x:t$), or a function application.

$$\begin{array}{lcl} V & ::= & x:t \mid f(V_1,\ldots,V_n) \\ C & ::= & V_1 = V_2 \mid \neg C \mid r(V_1,\ldots,V_n) \\ E & ::= & C \mid E_1 \wedge \cdots \wedge E_n \mid E_1 \vee \cdots \vee E_n \\ P & ::= & E \mid \forall x:t.\, P \mid \exists x:t.\, P \end{array}$$

We specify a set of candidate formulas by a triple (T,k,d). The *template* T is simply a formula P with a wildcard for the quantifier-free expression E. We define TemplateSpace(T,k,d) to be the set of formulas that match T when the wildcard is instantiated with any quantifier-free expression E that has at most k terms and a conjunction-disjunction tree of depth at most d. For instance, an expression of the form $c_1 \vee c_2 \vee c_3 \vee c_4$ has depth 1, while $(c_1 \vee c_2) \wedge (c_3 \vee c_4)$ has depth 2.

Hence, a valid candidate space might be defined by,

$$T = \forall r : \text{round}, v : \text{value}.\ \exists q : \text{quorum}.\ \forall n_1, n_2 : \text{node}.\ *$$

with $k = 3$, and $d = 1$, which would contain all invariants with the quantifiers in T and disjunctions of up to 3 terms. SWISS

expects the user to specify the maximum values of k, d, m (the maximum total number of quantified variables) and q (the maximum number of *existentially* quantified variables). The user must also specify a quantifier nesting order for the types defined by the protocols: a fixed nesting order is required so that the resulting verification conditions remain in EPR.

To enumerate the formulas in TemplateSpace(T,k,d), we first enumerate a set of possible terms, $\mathcal{C}$, and then arrange them in every possible tree shape. While succinct to describe, the size of the space grows exponentially, so we take steps to prune the space as rapidly as possible.

4.1.2 Symmetry-Breaking

As defined above, a candidate space contains many logically equivalent candidates, such as:

$$\forall r_1, r_2 : \text{round} \,.\, leq(r_1, r_2) \vee geq(r_1, r_2)$$
$$\forall r_1, r_2 : \text{round} \,.\, geq(r_1, r_2) \vee leq(r_1, r_2)$$
$$\forall r_1, r_2 : \text{round} \,.\, geq(r_2, r_1) \vee leq(r_2, r_1)$$

all identical up to term reordering and variable renaming.

When enumerating candidate formulas, SWISS aims to only consider one representative from each class of identical candidates. However, checking for such logical equivalence via SMT calls would be prohibitively expensive. Hence, we use a sound set of syntactic constraints to break these symmetries far more efficiently.

Specifically, SWISS assigns an arbitrary ordering to each possible base term (e.g., *leq*) and will only produce formulas where, within any conjunction or disjunction, the terms are in increasing order. SWISS also produces formulas such that for any set of quantified variables which are interchangeable (e.g., r_1 and r_2 in the example above), their first appearances are in increasing order of quantifier nesting index.

These two steps efficiently break symmetries arising from term ordering and variable permutation. In practice, they reduce the size of the search space by over two orders of magnitude (§5).

4.2 Filtering Based on Counterexamples

As explained in §2.4, when a candidate invariant P fails a verification condition, we can extract a counterexample, specifically a concrete model where the condition failed. For example, if $INIT \implies P$ fails to hold, then we can extract a concrete model M such that $M \models INIT \wedge \neg P$; i.e., the initial conditions hold for M but P does not. Since the initial conditions hold on M, if any other formula P' fails to hold for M, then it cannot be an invariant either. SWISS exploits this observation by remembering the models from failed VC checks and using those models to quickly rule out future candidates.

More technically, we define a *counterexample filter* (*cex*) as a model or a pair of models which demonstrate that a given candidate P fails to be an inductive invariant. We consider three types of counterexample filters:

$$cex \;::=\; \mathsf{True}(M) \mid \mathsf{False}(M) \mid \mathsf{Transition}(M, M')$$

A candidate P *passes* a filter *cex* if one of the following holds.

- $cex = \mathsf{True}(M)$ and $M \models P$; i.e., all valid invariants should evaluate to true on M.
- $cex = \mathsf{False}(M)$ and $M \models \neg P$.
- $cex = \mathsf{Transition}(M, M')$ and $(M, M') \models P \implies P'$.

The last is equivalent to: P passes $\mathsf{False}(M)$ *or* $\mathsf{True}(M')$.

We construct counterexample filters based on the model(s) returned when a candidate P fails a VC check. The specific type of filter constructed depends on which type of check fails. Our construction guarantees that **(i)** P does not pass the filter *cex*, and **(ii)** any formula P' which *does* pass the same verification condition *will* pass the filter *cex*. The three possible constructions are as follows.

- A failed verification condition of the form $A \implies P$ yields a model M such that $M \models A \wedge \neg P$, so we produce a counterexample filter $\mathsf{True}(M)$. In other words, P does not evaluate to *true* on M; but any valid invariant should.
- A failed verification condition of the form $A \wedge P \implies B'$ yields models M and M' such that $(M, M') \models A \wedge P \wedge \neg B'$, which gives a counterexample filter $\mathsf{False}(M)$ (since the model represents a state M that P failed to reject and that all valid invariants ought to reject).
- A failed verification condition of the form $A \wedge P \implies P'$ yields models M and M' such that $(M, M') \models A \wedge P \wedge \neg P'$, which gives a counterexample filter $\mathsf{Transition}(M, M')$.

Essentially, these counterexample filters allow us to learn from each type of failed induction check.

Efficient Implementation. Counterexample filtering is only useful if we can apply our filters faster than executing the original VC checks. Hence, whenever we add a new model M to our set of counterexample filters, we precompute the evaluation of each term $c \in \mathcal{C}$ on M. $\mathcal{C}$ is the set of base terms (§4.1.1) produced for every possible instantiation of the quantifier variables in the template T within the model M. The evaluations are saved in a bitstring, so when we encounter a new candidate Q, the evaluation of Q on M is computed quickly with bitwise operations. As a result, counterexample filters are orders of magnitude faster than VC checks (§5).

4.3 Checking Verification Conditions

We encode our verification conditions into SMT formulas in a manner similar to prior work [42]. The encoding ensures that it is decidable to evaluate the validity of the formulas. Our encoding conforms to the standardized `smtlib2` format, although for better performance, our implementation includes the SMT solver as a library so that it can directly invoke its API rather than communicating via files.

In practice, we find that thanks to the community's large effort invested in optimizing SMT solvers, our validity queries are not just decidable, but *rapidly* decidable, typically in tens of milliseconds. However, our initial experiments found occasional outliers that consumed minutes, or more. Since a SWISS run may perform thousands of SMT calls, these outliers become an issue. After some iteration, we settled on a routine

where we retry a problem instance with a different SMT solver if our first attempt exceeds a given time limit (§5.1). If the SMT instance continues to time out, we skip the invariant in question. While not a perfect solution, we found that this routine satisfactorily avoids stalling SWISS's execution.

4.4 Filtering Redundant Invariants

Since the Breadth algorithm searches rather indiscriminately for any possible invariant, it can easily waste time on redundant invariants, i.e., invariants already implied by previously established invariants.

To efficiently prune many such redundant invariants, SWISS includes a rapid FastImplies filter to see if an existing invariant Q already implies a candidate P. For correctness, FastImplies(Q,P) can return *true* only when $Q \implies P$. Our implementation of FastImplies(Q,P) always detects the case where Q and P can be written as,

$$Q = \forall^*\exists^* \ldots\ (a_1 \vee \cdots \vee a_j)$$
$$P = \forall^*\exists^* \ldots\ (b_1 \vee \cdots \vee b_k)$$

where $a_1,\ldots,a_j$ is a subsequence of $b_1,\ldots,b_k$ up to variable renaming. This condition certainly ensures that $Q \implies P$, meaning P is redundant if Q is already known to be invariant. Our implementation also has limited support for substituting universally-quantified variables for existential ones.

Efficient Implementation. To implement this filter efficiently, we store a set of sequences representing known invariants and query if any sequence in this set is a subsequence of a candidate sequence. We use a trie [10] for efficient queries.

4.5 Additional Optimizations

4.5.1 Minimizing Models

When extracting a counterexample from a failed VC check, we can either take the first model produced by the SMT solver, whatever it may be, or we can try to make the model as small as possible. Smaller models are faster to evaluate with our filters, but come at the cost of additional SMT queries.

In our *minimal-models* optimization, we always attempt to find a *minimal* model that satisfies a given SMT query, i.e., a model where there is no other satisfying model with a smaller domain. To bound the cost of the SMT queries made to check minimality, we apply an aggressive time limit to each query.

This is an essential optimization for complex protocols; without it, SWISS takes significantly longer (§5.3.5).

4.5.2 Parallelism

Since our algorithm is based on exhaustive search, we expect it to be parallelizable. To parallelize across n threads we do the following: for each run of either the Breadth or Finisher algorithm, we split the space ($\mathcal{B}$ or $\mathcal{F}$) into n parts, each randomly permuted. The random permutation attempts to ensure that each thread has a roughly equal amount of work. We then run our algorithm on each partition independently and combine the results. In the future, we plan to explore a more complex approach in which the threads communicate at a finer granularity, e.g., sharing counterexample filters and invariants as they are discovered.

4.6 Failed Optimizations

In the interest of full disclosure, and to save others unnecessary work, in Appendix A we briefly summarize three optimizations we implemented and evaluated, only to find that they provide little benefit or actively hurt performance.

5 Evaluation

Our evaluation seeks to answer two key questions:

- How does SWISS compare to prior work (§5.2)?
- How effective are SWISS's various optimizations (§5.3)?

5.1 Experimental Setup and Implementation Details

Benchmark Suite. To evaluate SWISS, we apply it to a test suite of well-established protocols from three sources: **(i)** the I4 benchmarks [34], **(ii)** the benchmarks from Koenig et al.'s work on first-order-logic separators [27] (henceforth, FOL), and **(iii)** six Paxos variants developed by Padon et al. [41].

Setup. All experiments are conducted on 8-core machines with Intel i9-9900K CPU processors at 3.60GHz, with 125GB of memory, running Ubuntu 19.10, and using a timeout of 6 hours. Unless specified, we run SWISS with 8 threads using automatic template generation (§4.1.1) and the following optimizations from §4: **(i)** symmetry breaking, **(ii)** cex filtering, **(iii)** hybrid SMT solvers, **(iv)** filtering redundant invariants, **(v)** model minimization, and **(vi)** parallelism. For template auto-generation, by default we use parameters (d,k,q,m) (§4.1.1) of $(1,3,1,5)$ for Breadth and $(2,6,1,6)$ for Finisher,

However, these choices were not ideal for all benchmarks. In particular, if we found that a benchmark completed in under six hours without proving the safety condition, we enlarged the $\mathcal{F}$ space by increasing k, and then increasing q. On the other hand, if a benchmark timed out after six hours without completing the breadth phase, we made the $\mathcal{B}$ space smaller by decreasing its parameters. See §5.2.1.

Implementation. We implement SWISS in approximately 14,000 lines of C++ code. To check the verification conditions, we use the Z3 SMT solver [11] version 4.8.8 with default settings. If Z3 times out after 45 seconds, then we use the CVC4 [2] SMT solver version 1.8-prerelease with the `--finite-model-find` option, an option tuned for finding finite models [44]. We found that for some satisfiable instances, Z3 would often take a very long time to return a model, whereas CVC4, *with this specific option*, was much more efficient. When applying model minimization, we set a time limit of 45 seconds; if the time limit is reached, we use the current partially minimized model.

5.2 Top-Level Protocol Results

Table 1 summarizes the results of running SWISS on our benchmark suite. It also compares SWISS's performance with that of the two most closely related systems (see §6 for details),

Source	Benchmark	∃?	I4 [34]	FOL [27]	SWISS	Partial	t_B	t_F	n_B	m_n	m_{n-1}
§2.2	sdl		10 s.	7 s.	5 s.		2 s.	2 s.	2	5	2
[34]	ring-election		3 s.	16 s.	11 s.		11 s.	-	3	3	3
	learning-switch-ternary		8 s.		195 s.		195 s.	-	2	3	3
	lock-server-sync		0.1 s.	1 s.	0.2 s.		0.2 s.	-	1	2	
	two-phase-commit		2 s.	15 s.	6 s.		6 s.	-	1	3	3
	chain		17 s.			(6 / 7)				7	4
	chord		506 s.	7 s.		(7 / 10)				6	4
	distributed-lock		131 s.			(1 / 4)				12	12
[27]	toy-consensus-forall			4 s.	3 s.		3 s.	-	2	3	3
	consensus-forall			1047 s.	29 s.		29 s.	-	3	3	3
	consensus-wo-decide			23 s.	18 s.		18 s.	-	3	3	3
	learning-switch-quad				959 s.		223 s.	736 s.	2	4	4
	lock-server-async		0.4 s.	2 s.	3684 s.		1 s.	3682 s.	1	8	
	sharded-kv		1.0 s.	7 s.	4024 s.		2 s.	4021 s.	2	8	2
	ticket			60 s.		(5 / 9)				6	6
	toy-consensus-epr	✓		22 s.	2 s.		2 s.	-	2	3	3
	consensus-epr	✓		377 s.	20 s.		20 s.	-	3	3	3
	client-server-ae	✓		303 s.	3 s.		2 s.	0.5 s.	2	3	
	client-server-db-ae	✓		2739 s.	24 s.		21 s.	2 s.	4	3	3
	sharded-kv-no-lost-keys	✓		1 s.	0.9 s.		0.9 s.	-	1	2	
	hybrid-reliable-broadcast	✓		791 s.		(1 / 7)				7	6
[41]	paxos	✓			15950 s.		803 s.	15146 s.	3	6	3
	flexible-paxos	✓			18232 s.		239 s.	17993 s.	3	6	3
	multi-paxos	✓				(6 / 11)				6	4
	multi-paxos*	✓			984 s.		589 s.	395 s.	3	6	4
	fast-paxos	✓				(8 / 12)				8	8
	stoppable-paxos	✓				(6 / 14)				6	6
	vertical-paxos	✓				(14 / 17)				16	6

Table 1: **Comparison with Prior Work.** ∃? indicates that the human-written invariant uses an existential. The time for SWISS is broken down into the time for Breadth (t_B) and Finisher (t_F), with the latter omitted when synthesis succeeds during the Breadth phase. n_B denotes number of iterations of Breadth. Each SWISS result is the median of five runs. For benchmarks where SWISS times out after 6 hours, we report its partial success (§5.2.2) instead of time. Shaded boxes indicate the system did not produce any invariants. In the multi-paxos* benchmark, the user provides a correct template for SWISS as guidance. The columns m_n and m_{n-1}, discussed in §5.3.4, summarize stats related to the sizes of invariants.

namely the I4 [34] and FOL [27] tools. Below, we first discuss the comparative results, followed by SWISS-specific analysis.

5.2.1 Comparative Results

We analyze our protocol results by bucketing them into coarse-grained categories. Note that I4 can only generate invariants containing universal quantifiers, and hence cannot succeed for benchmarks that require existentially quantified invariants.

Paxos Variants. Unlike prior work, SWISS automatically finds all of the invariants for Paxos and Flexible Paxos, which were previously painstakingly constructed by hand [41]. Furthermore, it succeeds at Multi-Paxos if the user provides the correct templates; however, when we attempted to use the automatically-enumerated templates, we found that Breadth does not complete in time. All three protocols require Finisher to find an invariant with six terms and an existential.

Unfortunately, neither I4, FOL, nor SWISS can prove the safety of Fast Paxos [30], Stoppable Paxos [35], or Vertical Paxos [32]. Among the known, handwritten invariants, Fast Paxos and Vertical Paxos each have two 8-term invariants; Stoppable Paxos has three 6-term invariants. SWISS cannot currently synthesize invariants this large—indeed, as it stands, it would need several orders of magnitude more compute time to handle even a single 8-term invariant—and we did not observe it finding equivalent, smaller invariants for these protocols. However, Table 1 shows that SWISS is at least able to synthesize partial invariants equivalent to many of the handwritten ones (discussed in more detail below — §5.2.2).

Finally, we note one interesting occurrence during our preliminary testing: we initially found that SWISS succeeded on Fast Paxos far too quickly and with invariants that looked at a glance to be far too strong. This allowed us to identify a typo in the spec which caused an action to never be enabled.

Mutual-Exclusion Protocols. Protocols such as sdl, distributed-lock, lock-server-sync, lock-server-async, and sharded-kv have safety conditions asserting mutual-exclusion

properties, such as those of locks. The invariants for these protocols tend to be large conjuncts of smaller, *mutually* inductive invariants. SWISS struggles with these, since it is forced to discover the entire collection at once, as none are inductive individually. For some, Finisher succeeds by inferring all of these mutually inductive invariants as a single large invariant. For sharded-kv and lock-server-async, we needed to increase the maximum value of k from our default configuration. For distributed-lock, the invariant was simply too large for SWISS to find. FOL was also unable to solve this benchmark, although I4 solves it quite handily, as it was able to infer the mutually inductive invariants on a small finite instance of the problem. Finally, for sharded-kv-no-lost-keys we needed to use $q = 2$ to allow more existentials.

Learning Switch. We have two different benchmarks called "learning-switch," one from I4 and one from FOL. We found that unlike I4 and FOL, SWISS succeeds on learning-switch-quad quite handily. In learning-switch-ternary, we found that Breadth takes far too long in the default configuration, due in part to a large number of redundant invariants not filtered by our FastImplies test and thus requiring SMT calls to identify as redundant. We reduced the size of $\mathcal{B}$ by configuring $q = 0$ (i.e., searching for universal invariants only) for this one benchmark, allowing SWISS to also be able to complete it efficiently.

5.2.2 SWISS-Specific Analysis

Invariants. In some cases, SWISS uncovered invariants that were simpler than the ones written by the original researchers. For instance, for the ring-election protocol, SWISS's Breadth algorithm identifies three 3-term invariants (9 terms total) similar to the ones from prior work [42]; however, if we run Finisher on its own rather than the full SWISS algorithm, then it instead generates a single 5-term inductive invariant. For the Paxos protocol, we initially expected that the Breadth algorithm would need to run with $k = 5$; however, we found that SWISS succeeded even with Breadth at $k = 3$. This shows that mechanized search can find simpler invariants that may be missed by trained researchers.

Partial Progress. When an execution of SWISS exceeds its time limit, it still generates many invariants which may be useful. To measure how successful this "partial progress" is, we computed how many of the handwritten invariants SWISS finds. More precisely, we count how many of these invariants are logical implications of SWISS's invariants. These results are shown in Table 1. Notice that in many cases, SWISS finds a majority of the invariants; e.g., for our largest benchmark, Vertical Paxos, SWISS finds 79%. Furthermore, for the chain benchmark, which has two safety conditions (one for linearizability and one for atomicity), SWISS solves the latter.

We do not report any partially solved invariants for the other tools in our comparison. For one, the IC3 approach does not immediately allow extraction of any inductive invariants; a partially-completed IC3 execution would provide a different kind of information: constraints on the possible states of a system after a finite number of transitions. In the case of I4, we did not find any case where it constructed a nontrivial inductive invariant for any benchmark where it did not succeed.

5.3 Understanding SWISS

To better understand SWISS, we evaluate the effectiveness of our various design choices.

5.3.1 The Utility of Combining Breadth and Finisher

In §3, we argue that Breadth and Finisher target different kinds of invariants and hence are more effective when used together. To validate this claim, we re-ran SWISS on our benchmark suite, first using only Breadth, and then using only Finisher. In both cases, we limited their execution time to that taken by the full SWISS algorithm.

Compared to the 19 benchmarks solved by the full SWISS algorithm, the Breadth algorithm alone solves only 10, while Finisher alone solves only 7.

5.3.2 Execution Times of Breadth Versus Finisher

In Table 1, Breadth is fairly quick, always less than 15 minutes, compared to Finisher which can vary from quite quick (a few seconds) to several hours. In fact, this is not an accident and follows directly from the fact that the space $\mathcal{B}$ is *chosen* to be small, whereas Finisher is designed to keep going until it finds a solution or times out. Thus, the runtime of Finisher depends on how large that solution turns out to be.

5.3.3 Impact of User Guidance

While SWISS is designed to automatically search through the space of invariant templates, users can provide more specific guidance via one or more templates. For example, for Paxos, considering the Finisher portion, the user might suggest a template like (1) or (2) in Table 2. Compared to searching the automatically generated space of templates, this guidance cuts the search space by 75% and as a result, completes in 20 minutes, a 13.4× speedup.

Of course, the user may erroneously suggest a template not containing any useful invariant. To evaluate the impact of such a mistake, we ran SWISS on a template (3) of comparable size to the "correct" templates, as well as some which are much smaller (4-6). The results suggest the user can afford to make some incorrect guesses and still outperform the full auto mode.

Finally, template (7) is the largest template generated automatically with our default configuration. It takes significantly longer, indicating the importance of exploring templates in increasing size order when using auto mode.

5.3.4 Is it a Small World?

Our Small World Hypothesis holds that many of the protocols we care about can be solved using a sequence of invariants $I_1, \ldots, I_n$, which are *individually* concise. We analyze this hypothesis by measuring the size of the invariants in each of our benchmark protocols. This, in turn, helps us understand

#	Template	Inv?	Size	Time (s)
1	$\forall r_1, r_2 : \text{round}, v_1, v_2 : \text{value}, q_1 : \text{quorum}.\ \exists n_1 : \text{node}.\ *$	✓	232,460,599,445	2036
2	$\forall r_1, r_2 : \text{round}, v_1, v_2 : \text{value}.\ \exists q_1 : \text{quorum}.\ \forall n_1 : \text{node}.\ *$	✓	232,460,599,445	4072
3	$\forall r_1, r_2 : \text{round}, v_1, v_2 : \text{value}, q_1 : \text{quorum}.\ \forall n_1 : \text{node}.\ *$		232,460,599,445	3547
4	$\forall r_1 : \text{round}, v_1, v_2, v_3 : \text{value}.\ \exists n_1 : \text{node}.\ *$		26,863,311,982	408
5	$\forall r_1 : \text{round}, v_1 : \text{value}, n_1, n_2, n_3, n_4 : \text{node}.\ *$		76,397,976,796	1015
6	$\forall r_1, r_2 : \text{round}, v_1 : \text{value}, q_1, q_2 : \text{quorum}.\ \exists n_1 : \text{node}.\ *$		39,834,946,595	557
7	$\forall r_1, r_2 : \text{round}, v_1, v_2 : \text{value}, n_1, n_2 : \text{node}.\ *$		2,621,795,213,086	47231

Table 2: **User-Provided Templates.** Experiments running Finisher on Paxos with the given template as input. Each experiment starts with many invariants provided as input, simulating the scenario that Breadth has already completed. The 'Inv?' column indicates whether the template contains an invariant that proves the safety condition. Time is given in seconds. Each experiment is run to completion, exploring the entire space, even if an invariant is found.

which protocols SWISS will likely succeed at by the degree to which they meet the Small World Hypothesis.

For a given protocol, we examine an invariant $I = I_1 \wedge \cdots \wedge I_n$ which proves the desired safety condition, chosen so that $I_1 \wedge \ldots \wedge I_j$ is inductive relative to the safety condition (as in Claim 1). The m_n column in Table 1 gives the maximum size of any invariant out of $I_1, \ldots, I_n$. Here, the "size" of a predicate is measured as its number of terms.

We do not claim that our numbers are the minimal possible—we simply use the smallest out of any I that we know of. These may be from invariants synthesized by SWISS itself; in other cases, we use human-determined invariants.

We can see that 25 out of our 27 benchmarks have $m_n \leq 8$, and 22 of them have even fewer. There were two exceptions, distributed-lock ($m_n = 12$) and vertical-paxos ($m_n = 16$). These two invariants are much bigger than any single invariant we have seen SWISS synthesize.

However, it may be worth noting that these larger invariants are conjuncts of smaller, mutually inductive invariants. For example, the 16-term invariant of vertical-paxos is actually the conjunct of two 8-term invariants (although these invariants are still on the larger side). Thus, these protocols would score much better on a weaker Small World Hypothesis, one where mutual invariants were counted separately. Other approaches may be able to take advantage of this.

To more fully understand SWISS's behavior, we also measure m_{n-1}, the maximum number of terms among $I_1, \ldots, I_{n-1}$. This statistic is interesting here because for SWISS to succeed in its current form, these $n-1$ invariants must be in $\mathcal{B}$.

To rough approximation, we see that SWISS can succeed approximately when $m_n \leq 6$ and $m_{n-1} \leq 3$. In some cases, SWISS does go beyond: some benchmarks succeed with up to $m_n = 8$, and SWISS can solve multi-paxos (where $m_{n-1} = 4$) if the search space is restricted in other ways.

5.3.5 Impact of Filtering

Table 3 shows the impact of each of our filtering stages on Paxos. Counterexample filtering drastically decreases the number of candidates which require an SMT call (i.e., those remaining after FastImplies). However, a vast number of candidates (the "Symmetries" column) must be processed by a counterexample-filter. How fast is such filtering?

	Baseline	Sym.	Cex filters	FastImpl	Inv.
$\mathcal{B}$	$820 \cdot 10^6$	$3 \cdot 10^6$	911,275	2,250	801
$\mathcal{F}$	$99 \cdot 10^{12}$	$232 \cdot 10^9$	155	155	5

Table 3: **Winnowing the Paxos Search Space**. Number of candidate predicates that remain *after* a given SWISS feature is applied. Runs are with a single thread over a single template.

To evaluate filtering efficiency, we measure the total time spent on filtering versus SMT inductivity checks in a Paxos benchmark. We find that Finisher (using Template (1) of Table 2) performs 155 inductivity checks via SMT. The average SMT call takes 96.5 ms, with a median of 7 ms and a 95$^{\text{th}}$ percentile of 55 ms. In contrast, filtering a single candidate takes 74 *nanoseconds* on average. Notably, both measures are important characteristics of SWISS, as some workloads are dominated by filtering and others by SMT calls (Appendix B.2).

5.3.6 Additional Analysis

See Appendix B for further analysis of SWISS's performance, e.g., the impact of optimizations, parallelism, and SMT calls.

6 Related Work

6.1 Verifying Distributed Systems

The research community has long recognized the challenges of designing correct distributed systems. Manually written proofs [25, 31] and model checking [23, 26, 56] increase assurance, but struggle with practical distributed systems [4].

Recent work applies general-purpose software-verification tools to the verification of distributed systems [21, 46, 55]. These tools offer flexibility at the price of substantial human effort. For example, verifying Raft required over 50,000 lines of Coq proof for the protocol and its 520-line implementation [55], and Hawblitzel et al. used 12,000 lines of proof for the safety and liveness proofs of their Paxos protocol [21].

These human costs motivate the search for domain-specific languages (DSLs) and tools that reduce proof effort by re-

stricting the class of encodable systems [12, 13, 36, 53]. For example, tools based on the heard-of model [12, 13, 36] need a run-time system to bridge the gap between an asynchronous network and the synchronous semantics assumed by the verification tool, which can lead to performance bottlenecks. Similarly, pretend synchrony [53] precludes classic optimizations, e.g., request batching in Paxos implementations.

All the works above, even the DSLs, rely on the developer to intuit invariants, which can be hard even for experienced researchers [34]. Recent work tries to reduce this cost via a restricted logic (EPR – §2.4) which makes invariant checking decidable; i.e., given a correct invariant, no further human work is needed. Even within EPR, *finding* the invariant remains undecidable [40], so Padon et al.'s IVy tool [42] interactively aids the developer: IVy iteratively checks if a candidate invariant is inductive. If not, IVy presents a concrete counterexample, and the developer strengthens the candidate to eliminate it. This repeats until she derives an inductive invariant.

In contrast, Ma et al.'s I4 tool [34] aims to be fully automatic. I4 first runs a custom model checker [19] on an artificially small example of the protocol (e.g., with two nodes) to produce an invariant for the small system. I4 then attempts to generalize the invariant to the unbounded setting. When it succeeds, I4 requires no human intervention. In practice, Ma et al. report manually specifying concrete bounds for all of their benchmarks (to avoid exhaustive parameter searches) and concretizations of certain variables for several benchmarks. Ultimately, I4 is limited by the abilities of its model checker, which does not support existential quantifiers (which §5.2 shows rules out a wide swath of protocols), and is unable to scale to more complex protocols like Paxos. In such cases, the developer is left with little recourse. However, I4 is frequently faster than SWISS and able to synthesize larger invariants for protocols where universally quantified invariants exist.

In recent work, Koenig et al. (FOL) [27] develop an algorithm capable of synthesizing invariants containing existentials. Their algorithm relies on the IC3/PDR algorithm [6, 14] for constructing invariants incrementally. Like SWISS, it iteratively produces counterexamples, but it uses those counterexamples as constraints in a SAT encoding of predicates to be synthesized. SWISS verifies protocols, like Paxos, that FOL does not and verifies some faster than FOL. The reverse is also true, suggesting some complementarity of the approaches.

6.2 General-Purpose Invariant Synthesis

Extensive research [7–9, 15, 17, 18, 20, 37, 39, 49, 57] studies loop-invariant inference for proving program correctness, but this remains challenging. Most approaches are limited to single-loop programs; only a few handle multiple loops or existential invariants. Approaches include abstract interpretation [8], interpolation [37], IC3 [16, 17], templates and constraint solvers [20], counterexample-guided invariant generation (CEGIR) [18, 39, 47], trace analysis [9, 15], and machine learning [48, 49, 57].

More closely related work uses templates [7, 20] to restrict the search to invariants of a given shape. In contrast to these approaches, we automatically construct a large set of templates and search for invariants of larger sizes. CEGIR approaches [18, 39, 47] use enumeration and exploit the fact that guessing a candidate and checking if it is invariant is easier than inferring a loop invariant directly from code. They often employ dynamic analyses to infer candidates from execution traces and use a verifier to check invariant validity. The idea of learning from counterexamples has also been applied to program synthesis in the form of counterexample-guided inductive synthesis (CEGIS) [51], where the synthesizer generates a candidate program and the verifier uses the failed cases to prune the search space. SWISS's approach is inspired by techniques from search-based program synthesis [1] and the CEGIS framework. Although CEGIS is a general framework, it cannot be used as a black-box since it requires a custom synthesizer, verifier, and learner for each domain. SWISS differs from prior approaches in how it uses the counterexamples to prune the search space (§4.2) and how it applies a CEGIS-style approach to infer more complex invariants than prior work.

Program sketching [50] allows a programmer to sketch a program, i.e., write a program with "holes." A synthesizer fills the holes such that a specification is satisfied. In SWISS, the user can similarly provide templates to restrict the search, but even if such a template is not provided, SWISS can automatically generate a set of templates and search all of them.

7 Conclusions and Future Work

We explore the hypothesis that the safety of most distributed systems can be proven via relatively small invariants (or conjunctions thereof), using our system SWISS, which incorporates novel optimizations to efficiently search the space of candidate invariants. We find that in many cases our hypothesis holds, and SWISS is able to automatically prove their safety, including several, such as Paxos, beyond the reach of prior work. Our results leave open the question of inferring large, mutually inductive invariants. They also illustrate that SWISS and its most recent predecessors often have complementary coverage of the benchmarks. Exploring ways to combine the strengths of each is an intriguing direction for future work.

8 Acknowledgements

We thank Jay Lorch, Jon Howell, the NSDI reviewers, and our shepherd, Siddhartha Sen, for useful paper feedback. We also thank Jean Yang and Mickaël Laurent for many helpful conversations and early explorations. We thank the Pittsburgh Supercomputing Center for providing computation resources.

This work was funded in part by the Alfred P. Sloan Foundation, a Google Faculty Fellowship, the NSF/VMware Partnership on Software Defined Infrastructure as a Foundation for Clean-Slate Computing Security (SDI-CSCS) program under Award No. CNS-1700521, and the NSF under grant CCF-2015445.

References

[1] ALUR, R., SINGH, R., FISMAN, D., AND SOLAR-LEZAMA, A. Search-based program synthesis. *Commun. ACM 61*, 12 (2018), 84–93.

[2] BARRETT, C., CONWAY, C. L., DETERS, M., HADAREAN, L., JOVANOVI'C, D., KING, T., REYNOLDS, A., AND TINELLI, C. CVC4. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806 of *Lecture Notes in Computer Science*, Springer, pp. 171–177.

[3] BOLOSKY, W. J., DOUCEUR, J. R., AND HOWELL, J. The Farsite project: A retrospective. *SIGOPS Oper. Syst. Rev. 41*, 2 (Apr. 2007), 1726.

[4] BOLOSKY, W. J., DOUCEUR, J. R., AND HOWELL, J. The Farsite project: a retrospective. *ACM SIGOPS Operating Systems Review 41 (2)* (April 2007).

[5] BORNHOLT, J., AND TORLAK, E. Synthesizing memory models from framework sketches and litmus tests. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (2017).

[6] BRADLEY, A. R. SAT-based model checking without unrolling. In *Proceedings of Verification, Model Checking, and Abstract Interpretation (VMCAI)* (2011).

[7] COLÓN, M., SANKARANARAYANAN, S., AND SIPMA, H. Linear invariant generation using non-linear constraint solving. In *Proceedings of the International Conference on Computer Aided Verification (CAV)* (2003), W. A. H. Jr. and F. Somenzi, Eds., vol. 2725 of *Lecture Notes in Computer Science*, Springer, pp. 420–432.

[8] COUSOT, P., AND HALBWACHS, N. Automatic discovery of linear restraints among variables of a program. In *Proceedings on the ACM Symposium on Principles of Programming Languages (POPL)* (1978), A. V. Aho, S. N. Zilles, and T. G. Szymanski, Eds., ACM Press, pp. 84–96.

[9] DANESE, A., PICCOLBONI, L., AND PRAVADELLI, G. A parallelizable approach for mining likely invariants. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis* (2015), CODES '15, IEEE Press, p. 193201.

[10] DE LA BRIANDAIS, R. File searching using variable length keys. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference* (New York, NY, USA, 1959), IRE-AIEE-ACM 59 (Western), Association for Computing Machinery, p. 295298.

[11] DE MOURA, L., AND BJØRNER, N. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2008), Springer, pp. 337–340.

[12] DRĂŹGOI, C., HENZINGER, T. A., VEITH, H., WIDDER, J., AND ZUFFEREY, D. A logic-based framework for verifying consensus algorithms. In *Proceedings of Computer Aided Verification (CAV)* (2014).

[13] DRĂŹGOI, C., HENZINGER, T. A., AND ZUFFEREY, D. PSync: A partially synchronous language for fault-tolerant distributed algorithms. In *ACM Symposium on Principles of Programming Languages (POPL)* (2016).

[14] EÉN, N., MISHCHENKO, A., AND BRAYTON, R. Efficient implementation of property directed reachability. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD)* (2011).

[15] ERNST, M. D., PERKINS, J. H., GUO, P. J., MCCAMANT, S., PACHECO, C., TSCHANTZ, M. S., AND XIAO, C. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program. 69*, 13 (Dec. 2007), 3545.

[16] EZUDHEEN, P., NEIDER, D., D'SOUZA, D., GARG, P., AND MADHUSUDAN, P. Horn-ICE learning for synthesizing invariants and contracts. *Proc. ACM Program. Lang. 2*, OOPSLA (2018), 131:1–131:25.

[17] GARG, P., LÖDING, C., MADHUSUDAN, P., AND NEIDER, D. ICE: A robust framework for learning invariants. In *Proceedings of the International Conference on Computer Aided Verification (CAV)* (2014), A. Biere and R. Bloem, Eds., vol. 8559 of *Lecture Notes in Computer Science*, Springer, pp. 69–87.

[18] GARG, P., NEIDER, D., MADHUSUDAN, P., AND ROTH, D. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (2016), R. Bodík and R. Majumdar, Eds., ACM, pp. 499–512.

[19] GOEL, A., AND SAKALLAH, K. A. Model checking of verilog RTL using IC3 with syntax-guided abstraction. In *Proceedings of the NASA Formal Methods Symposium (NFM)* (2019), J. M. Badger and K. Y. Rozier, Eds., vol. 11460 of *Lecture Notes in Computer Science*, Springer, pp. 166–185.

[20] GUPTA, A., AND RYBALCHENKO, A. InvGen: An efficient invariant generator. In *Proceedings of the International Conference on Computer Aided Verification (CAV)* (2009), A. Bouajjani and O. Maler, Eds., vol. 5643 of *Lecture Notes in Computer Science*, Springer, pp. 634–640.

[21] HAWBLITZEL, C., HOWELL, J., KAPRITSOS, M., LORCH, J. R., PARNO, B., ROBERTS, M. L., SETTY, S., AND ZILL, B. IronFleet: Proving safety and liveness of practical distributed systems. *Commun. ACM 60*, 7 (June 2017), 8392.

[22] HOWARD, H., MALKHI, D., AND SPIEGELMAN, A. Flexible Paxos: Quorum intersection revisited, 2016.

[23] JONES, E. Model checking a Paxos implementation. http://www.evanjones.ca/model-checking-paxos.html, 2009.

[24] JOSHI, R., LAMPORT, L., MATTHEWS, J., TASIRAN, S., TUTTLE, M., AND YU, Y. Checking cache-coherence protocols with TLA+. *Journal of Formal Methods in System Design 22* (March 2003), 125–131.

[25] KELLOMÄKI, P. An annotated specification of the consensus protocol of Paxos using superposition in PVS. Tech. Rep. 36, Tampere University of Technology, 2004.

[26] KILLIAN, C. E., ANDERSON, J. W., BRAUD, R., JHALA, R., AND VAHDAT, A. M. Mace: Language support for building distributed systems. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (2007).

[27] KOENIG, J. R., PADON, O., IMMERMAN, N., AND AIKEN, A. First-order quantified separators. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)* (2020), A. F. Donaldson and E. Torlak, Eds., ACM, pp. 703–717.

[28] LAMPORT, L. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems 16*, 3 (May 1994).

[29] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133169.

[30] LAMPORT, L. Fast Paxos. *Distributed Computing 19* (October 2006), 79–103.

[31] LAMPORT, L. Byzantizing Paxos by refinement. In *Proceedings of the International Conference on Distributed Computing (DISC)* (2011).

[32] LAMPORT, L., MALKHI, D., AND ZHOU, L. Vertical paxos and primary-backup replication. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)* (2009), S. Tirthapura and L. Alvisi, Eds., ACM, pp. 312–313.

[33] LEWIS, H. R. Complexity results for classes of quantificational formulas. *J. Comput. Syst. Sci. 21* (1980), 317–353.

[34] MA, H., GOEL, A., JEANNIN, J., KAPRITSOS, M., KASIKCI, B., AND SAKALLAH, K. A. I4: incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the ACM Symposium on Operating Systems Principles, (SOSP)* (2019), T. Brecht and C. Williamson, Eds., ACM, pp. 370–384.

[35] MALKHI, D., LAMPORT, L., AND ZHOU, L. Stoppable Paxos. Tech. Rep. MSR-TR-2008-192, April 2008.

[36] MARIC, O., SPRENGER, C., AND BASIN, D. A. Cutoff bounds for consensus algorithms. In *Proceedings of Computer Aided Verification (CAV)* (2017).

[37] MCMILLAN, K. L. Quantified invariant generation using an interpolating saturation prover. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2008), C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963 of *Lecture Notes in Computer Science*, Springer, pp. 413–427.

[38] NEWCOMBE, C., RATH, T., ZHANG, F., MUNTEANU, B., BROOKER, M., AND DEARDEUFF, M. How Amazon web services uses formal methods. *Commun. ACM 58*, 4 (Mar. 2015), 6673.

[39] NGUYEN, T., ANTONOPOULOS, T., RUEF, A., AND HICKS, M. Counterexample-guided approach to finding numerical invariants. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)* (2017), E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds., ACM, pp. 605–615.

[40] PADON, O., IMMERMAN, N., SHOHAM, S., KARBYSHEV, A., AND SAGIV, M. Decidability of inferring inductive invariants. *SIGPLAN Not. 51*, 1 (Jan. 2016), 217231.

[41] PADON, O., LOSA, G., SAGIV, M., AND SHOHAM, S. Paxos made EPR: Decidable reasoning about distributed protocols. *Proc. ACM Program. Lang. 1*, OOPSLA (Oct. 2017).

[42] PADON, O., MCMILLAN, K. L., PANDA, A., SAGIV, M., AND SHOHAM, S. Ivy: Safety verification by interactive generalization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2016), Association for Computing Machinery, p. 614630.

[43] PISKAC, R., DE MOURA, L., AND BJRNER, N. Deciding effectively propositional logic with equality. Tech. Rep. MSR-TR-2008-181, December 2008.

[44] REYNOLDS, A., TINELLI, C., GOEL, A., AND KRSTIC, S. Finite model finding in SMT. In *Proceedings of the International Conference on Computer Aided Verification (CAV)* (2013), Springer, pp. 640–655.

[45] SCHIPER, N., RAHLI, V., RENESSE, R. V., BICKFORD, M., AND CONSTABLE, R. L. Developing correctly replicated databases using formal tools. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2014), IEEE Computer Society, p. 395406.

[46] SCHIPER, N., RAHLI, V., VAN RENESSE, R., BICKFORD, M., AND CONSTABLE, R. Developing correctly replicated databases using formal tools. In *Proceedings of the IEEE/IFIP Conference on Dependable Systems and Networks (DSN)* (June 2014).

[47] SHARMA, R., GUPTA, S., HARIHARAN, B., AIKEN, A., LIANG, P., AND NORI, A. V. A data driven approach for algebraic loop invariants. In *Proceedings of the European Symposium on Programming Languages and Systems (ESOP)* (2013), M. Felleisen and P. Gardner, Eds., vol. 7792 of *Lecture Notes in Computer Science*, Springer, pp. 574–592.

[48] SI, X., DAI, H., RAGHOTHAMAN, M., NAIK, M., AND SONG, L. Learning loop invariants for program verification. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada* (2018), S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., pp. 7762–7773.

[49] SI, X., NAIK, A., DAI, H., NAIK, M., AND SONG, L. Code2Inv: A deep learning framework for program verification. In *Proceedings of the International Conference on Computer Aided Verification (CAV)* (2020), S. K. Lahiri and C. Wang, Eds., vol. 12225 of *Lecture Notes in Computer Science*, Springer, pp. 151–164.

[50] SOLAR-LEZAMA, A. Program sketching. *Int. J. Softw. Tools Technol. Transf. 15*, 5-6 (2013), 475–495.

[51] SOLAR-LEZAMA, A., TANCAU, L., BODÍK, R., SESHIA, S. A., AND SARASWAT, V. A. Combinatorial sketching for finite programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2006), J. P. Shen and M. Martonosi, Eds., ACM, pp. 404–415.

[52] TRAKHTENBROT, B. The impossibility of an algorithm for the decidability problem on finite classes. In *Proceedings of the USSR Academy of Sciences* (1950).

[53] V. GLEISSENTHALL, K., KICI, R. G., BAKST, A., STEFAN, D., AND JHALA, R. Pretend synchrony: Synchronous verification of asynchronous distributed programs. In *ACM Symposium on Principles of Programming Languages (POPL)* (2019).

[54] WILCOX, J. R., WOOS, D., PANCHEKHA, P., TATLOCK, Z., WANG, X., ERNST, M. D., AND ANDERSON, T. Verdi: A framework for implementing and formally verifying distributed systems. *SIGPLAN Not. 50*, 6 (June 2015), 357368.

[55] WOOS, D., WILCOX, J. R., ANTON, S., TATLOCK, Z., ERNST, M. D., AND ANDERSON, T. Planning for change in a formal verification of the raft consensus protocol. In *ACM Conference on Certified Programs and Proofs (CPP)* (Jan. 2016).

[56] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (April 2009).

[57] YAO, J., RYAN, G., WONG, J., JANA, S., AND GU, R. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2020), PLDI 2020, Association for Computing Machinery, p. 106120.

[58] YU, Y. Using formal specifications to monitor and guide simulation: Verifying the cache coherence engine of the Alpha 21364 microprocessor. In *Proceedings of the IEEE International Workshop on Microprocessor Test and Verification (MTV)* (2002), Institute of Electrical and Electronics Engineers, Inc.

A Failed Optimizations

In the interest of full disclosure, and to save others unnecessary work, we briefly summarize three optimizations we implemented and evaluated, only to find that they provide little benefit or actively hurt performance.

A.1 Formula Synthesis

SWISS's current implementation explicitly enumerates and evaluates candidate invariants. Initially, however, we adopted a strategy from a prior system, MemSynth [5], which synthesizes memory models from a small collection of examples. Instead of explicitly enumerating formulas, MemSynth encodes the desired shape of candidate formulas as constraints on its SMT queries. In our context, this means creating an SMT query that says, "Find a formula that satisfies this template and complies with our accumulated counterexamples."

However, our early experiments showed that synthesizing the formula via a solver was considerably slower than our combination of a custom enumerator and counterexample filters.

A.2 Bounded Model Checking

Prior work on finding invariants for distributed systems [42], found some benefit from using bounded model checking to try to quickly rule out candidate invariants. In our context, this means checking not just the condition $INIT \implies P$, but whether there are any violations of P in states reachable after taking a fixed number of steps from an initial state. If such a violation existed on a model M, we can use the counterexample filter $\mathsf{True}(M)$. Likewise, to build False filters, we consider states a fixed number of steps away from violating safety.

The hypothesis for this optimization is that it would produce more and "higher quality" filters to rule out future candidates. To pay off, these savings must offset the cost of the additional SMT calls that compute the bounded model checks. Sadly, this optimization rarely boosts performance significantly (§5.3.5).

A.3 Aggressively Accumulating Invariants

As it executes, Breadth finds predicates that are invariant with respect to the input invariants $I_1, \ldots, I_n$; these new invariants are fed into the next iteration of Breadth. This suggests an obvious improvement: treat newly found invariants as input invariants immediately in order to uncover even more invariants than the ones Breadth is guaranteed to find, leading to fewer total loops. We call this variation BreadthAccumulative.

However, this variant introduces several complications. Most critically, it interferes with the FastImplies optimization. For example, suppose we process predicates $f, g, h_1, \ldots, h_n$ in order and **(i)** g is invariant; **(ii)** f is invariant with respect to g; and **(iii)** $\mathsf{FastImplies}(f, h_i)$ holds for all i. BreadthAccumulative, would pass over f (not invariant), then find and add g, causing all h_i to become invariant. Since the h_i are not filtered out by the FastImplies check, BreadthAccumulative's aggressive addition of the h_i causes the number of invariants to explode. In contrast, Breadth would only add g at the end; then in the next call to Breadth, f would be added to allInv, excluding the h_i via FastImplies.

To prevent BreadthAccumulative from adding such spurious invariants, we added a *strengthening* step, where the first h_i found would be strengthened to f. However, strengthening comes at a cost, and in the end, we found the BreadthAccumulative optimization to be unhelpful (Figure B.2).

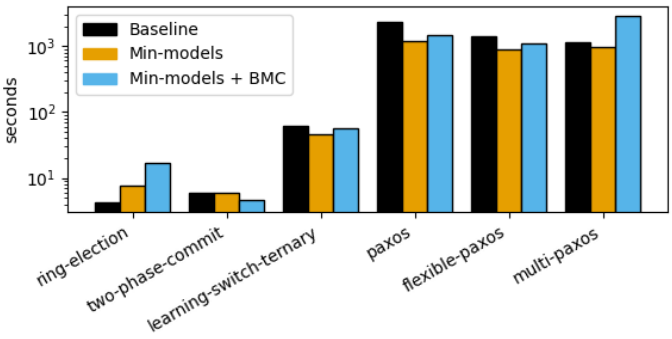

Figure 5: **Optimization Impact**. We evaluate our model-minimization and bounded-model-checking optimizations. To save time, these experiments use pre-specified templates rather than template auto-generation. Note the log scale of the y-axis.

B Additional Evaluation

In this section, we provide some additional measurements of the impact of SWISS's design decisions.

B.1 Impact of Optimizations

Model Minimization. To evaluate the effectiveness of model minimization (§A.2), we measure several benchmarks with and without it (Figure 5). While it adds some overhead for simple protocols, it helps significantly for more complex protocols; in the best case, we found that it improved the Flexible-Paxos experiment by 1.6×.

Bounded Model Checking. For our BMC variant (§A.2), we again ran several experiments with and without it (Figure 5). However, the results show that the cost of the required SMT calls was not sufficiently offset by gains from "higher quality" filters. Hence we disable it in SWISS's default configuration.

B.2 Impact of Parallelism

To evaluate our parallelism strategy (§4.5.2), we measure runtimes with varying numbers of threads, studying Breadth and Finisher independently. For each run, we break down the running time of the longest-running thread in each iteration to see which components of the algorithm parallelize well. We run our experiments on the Paxos protocol.

For the purposes of this experiment, unlike in standard runs, we do not terminate the Finisher algorithm when it finds an invariant which proves the safety condition; instead, we let it search the entire search space $\mathcal{F}$. This removes variance

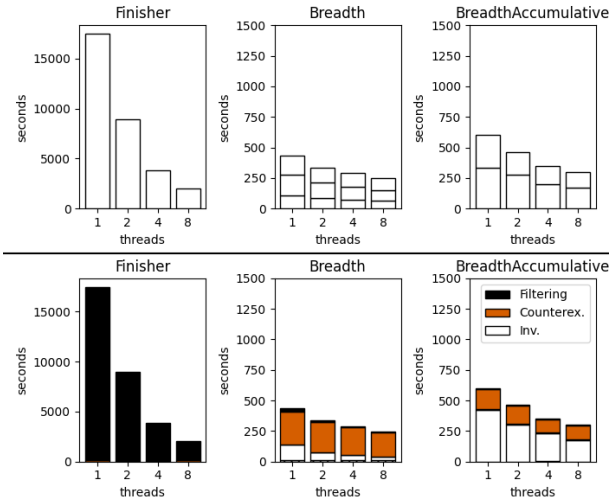

Figure 6: **Parallelism**. We measure time while varying the number of threads for Finisher, Breadth and BreadthAccumulative (§A.3) on Paxos. The top row shows the overall time, with Breadth broken down by iterations. Breadth terminates when the last (top) iteration fails to find any invariant, so it takes one less iteration than shown to find all invariants Breadth can find. In the bottom row, the runtimes are broken down into **(i)** filtering candidates with counterexamples, **(ii)** computing counterexamples of non-invariants, and **(iii)** processing candidates that are invariant.

from the random ordering of the search space, leading to more controlled experimental data.

Our results are shown in Figure 6. Finisher's runtime is dominated by enumerating and filtering, which splits fairly evenly across threads. Overall, SWISS on a single template is 2.0× faster with 2 threads than with 1 thread, and it is 8.6× faster with 8 threads than with 1.

Breadth, meanwhile, does not parallelize as well, since its runtime is dominated by time spent constructing counterexample filters via SMT calls, which is essentially a fixed cost per thread. At best, we saw a speedup of 1.7× with 8 threads.

We also evaluated BreadthAccumulative (§A.3) while varying the number of threads (Figure 6). Our hypothesis that BreadthAccumulative would require fewer iterations was confirmed: each run required only one iteration to find all invariants in $\mathcal{B}$ (plus one iteration to confirm no further invariants exist). By contrast, Breadth requires two iterations (plus one) on the same benchmark. However, BreadthAccumulative still performs worse than Breadth due its other costs (e.g., strengthening – §A.3).

B.3 Hard SMT instances

As described in §4.3, SMT queries are often rapid, but there are occasional outliers that slow down execution. To measure how problematic these outliers are, we measured the prevalence of hard instances, defined as any instance exceeding forty-five seconds and triggering our retry strategy. In particular, we measured the fraction of total computation time spent on these hard instances.

Among all protocols that SWISS was able to solve, this fraction was greatest for the paxos benchmark, which spent 10.2% of its computation time on hard instances, which accounted for 0.17% of its SMT instances.

However, among all protocols that SWISS was *not* able to solve, this fraction was greatest for the fast-paxos benchmark, which spent 98.7% of its computation time on hard instances, which accounted for 3.7% of its SMT instances. We currently do not have a good understanding of what makes this protocol's SMT instances difficult for SMT solvers, but the numbers suggest that improvement to SWISS's SMT strategy could make it much faster on harder protocols.