# Invariant Synthesis for Distributed Protocols using FO-ICE

CS598MP Fall 2021 Project Report

Parth Thakkar and Adithya Murali
{parthdt2, adithya5}@illinois.edu

May 2022

## 1 Introduction

In this work we study the problem of synthesizing invariants for proving the safety of distributed protocols. This problem has been explored in recent work Yao et al. [2021], Hance et al. [2021], Koenig et al. [2022], where protocols are described in a language such as Ivy Padon et al. [2016]. We note here that the problem of protocol verification is different from the problem of verifying a distributed system implementation.

### 1.1 Preliminaries

We now describe the components of a distributed protocol, which is described over a universe of nodes $\mathcal{N}$ participating in the protocol. A protocol description consists of a *global state* that takes values in a set $\mathcal{S}$ (which we also call states for convenience), and a set of *actions* $\mathcal{A} = \{a : \mathcal{S} \to \mathcal{S}\}$. An execution of a distributed system is a sequence of actions. We say that the distributed system *transitions* from $s_1$ to $s2$ on $a$ if $a(s_1) = s_2$.

Each action is often guarded by a *precondition* that determines if the action can be taken at the current global state. We only consider such well-formed executions according to the preconditions. Finally, the description also specifies a set of initial states $Init \subseteq \mathcal{S}$ such that well-formed executions always begin with an initial state. We refer to the set of all states that feature in any well-formed execution as the *reachable states*.

Figure 1 shows an example of a distributed lock protocol featured in the work in Hance et al. [2021].

Consider first-order logic over the universe of nodes where we model the global state using relations and functions. For example, for the protocol in Figure 1 we model the global state using a binary relation *message* and a unary relation *lock*. The safety specification for a distributed protocol is a first-order formula in this logic. Figure 2 shows the specification for the protocol in Figure 1.

A distributed protocol is said to be safe if every reachable state satisfies the safety property. Proving safety is typically achieved by finding an *invariant* such that: (1) the initial states satisfy the invariant (2) the invariant is preserved under states related by a single well-formed transition (3) the invariant implies the safety property.

### 1.2 Verification Conditions

We capture the above requirements in terms of *verification conditions* (VCs) and reduce safety verification to checking the validity of first-order formulae. We first extend the above signature with

**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: Example of a distributed protocol Hance et al. [2021]

**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: Example of a distributed protocol specification and invariants used to prove safety Hance et al. [2021]

an additional sort *State* to model states and extend the signature of relations and functions with an additional argument of the state sort. In the above example, the signature now consists of relations $message(S : State, src : node, dst : node)$ and $lock(S : State, n : node)$. The models of this extended signature can be seen as consisting of closed sub-universes corresponding to each state in the carrier set of the *State* sort. Each such sub-model represents a possible state of the distributed system.

The set of initial states is typically described by a formula, which we refer to as $Init(S)$. We denote the safety property by $Safe(S)$. Finally, we capture the transition effected by an action $a \in \mathcal{A}$ using a formula $Trans_a(S_1, S_2)$. The formula $Trans_a$ also checks that $S_1$ satisfies the precondition for $a$.

Let the invariant be denoted by $\varphi(S)$. The VCs can then be written as follows.

$$\forall S : State.\ Init(S) \implies \varphi(S) \qquad \text{(initial states satisfy invariant)}$$

$$\forall S_1, S_2 : State.\ (\varphi(S_1) \wedge Trans_a(S_1, S_2)) \implies \varphi(S_2)\ \text{, for every a } \in \mathcal{A}$$
(invariant is preserved under well-formed transitions)

$$\forall S : State.\ \varphi(S) \implies Safe(S) \qquad \text{(invariant implies the safety property)}$$

## 1.3 Synthesizing a Sequence of Invariants

In this work we study synthesizing invariants for proving the safety of distributed protocols. As we discuss in the sequel, we model this as a SyGuS Alur et al. [2015] problem, providing a grammar $\mathcal{G}$ from which invariants must be chosen such that the verification conditions in Section 1.2 are valid. Since the invariant for a distributed protocol is typically very large, we modify the problem to require the synthesis of several partial invariants $\varphi_1, \varphi_2, \ldots \varphi_n$ such that each $\varphi_i$ belongs to the language $L(\mathcal{G})$ and the conjunction $\bigwedge_{i=1}^{n} \varphi_i$ is the invariant that satisfies the verification conditions.

In this project we formulate algorithms for a sub-class of the above problem. Since we wish to synthesize these partial invariants one at a time, we demand that each partial invariant be provable as inductive (with respect to the protocol transitions) given the previously discovered partial invariants. We define this *sequence of inductive partial invariants* below.

**Definition 1 (Sequence of inductive partial invariants)** *Given a protocol description with the initial state condition Init, transitions $Trans_a$ for $a \in \mathcal{A}$ and the safety property Safe, a sequence of formulas $\varphi_1, \varphi_2 \ldots \varphi_n$ is said to be a sequence of inductive partial invariants iff:*

*(a)* $\forall S : State.\ Init(S) \implies \varphi_i(S)$ *for every* $1 \leq i \leq n$

*(b)* $\forall S_1, S_2 : State.\ \left(\bigwedge_{j=1}^{i-1} \varphi_j(S_1) \wedge \varphi_i(S_1) \wedge Trans_a(S_1, S_2)\right) \implies \varphi_i(S_2)$ *for every action* $a \in \mathcal{A}$, *for every* $1 < i \leq n$

*(c)* $\forall S : State.\ (\bigwedge i = 1^n \varphi_i(S)) \implies Safe(S)$

The second condition above expresses that every partial invariant must be inductive in the presence of previously discovered ones.

In the following sections we formulate algorithms to synthesize a sequence of inductive partial invariants to prove distributed protocols safe. We also restrict ourselves to scenarios where the partial invariants are universally quantified. Finally, we refer to partial invariants as invariants when it is clear from context.

# 2 FO-ICE

In this section we discuss our formulation of invariant synthesis for distributed protocols using counterexamples.

We use the ICE framework Garg et al. [2014], designing positive, negative, and implication counterexamples. Unlike prior work on ICE, our counterexamples are first-order models and we synthesize quantified first-order formulas. We therefore refer to our approach as FO-ICE.

Counterexamples are formulated in terms of the states of the distributed system. Recall that a state corresponds to a closed sub-model in terms of the extended signature introduced in Section 1.2. We denote counterexamples using symbols $C_1$, $C_2$, etc. Further, we denote the evaluation of a candidate invariant $\varphi$ on a state $S$ by $\varphi(S)$. We describe the three kinds of counterexamples below. Let $\varphi$ denote a candidate partial invariant.

**Positive counterexample** A positive counterexample consists of a single state $S$ such that $\mathit{Init}(S) \not\Rightarrow \varphi(S)$.

**Negative counterexample** A negative counterexample to consists of a single state $S$ such that $\varphi(S) \not\Rightarrow \mathit{Safe}(S)$.

**Implication counterexample** An implication counterexample consists of a pair of states $(S_1, S_2)$ such that $(\varphi(S_1) \wedge \mathit{Trans}_a(S_1, S_2)) \not\Rightarrow \varphi(S_2)$.

## 2.1 Synthesis Algorithm

In this section we describe our basic synthesis algorithm, and document additional changes or optimizations made to the algorithm in the following sections.

We describe the algorithm at a high-level, assuming a synthesis engine *ModelSynth* that can synthesize universally quantified formulas given counterexample models. We only use the knowledge here that *ModelSynth* returns a candidate invariant that eliminates all given counterexamples. We describe the how we realize this engine in Section 2.2.

Figure 3 contains the pseudocode for the FO-ICE algorithm. FO-ICE takes as input the description of the protocol, including the initial state condition, the transition formulae for each action, and the safety property. It also takes as input a grammar $\mathcal{G}$ that specifies the search space of possible invariants.

The algorithm maintains a sequence *Inv* of invariants (initially empty) and a set *Cex* of counterexamples (initially empty). At a general point in the algorithm, we have a sequence of inductive partial invariants *Inv* (see Definition 1) that are together not sufficient to prove safety and a set of various types of counterexamples *Cex* such that every partial invariant in *Inv* satisfies the counterexamples (i.e., is not eliminated by the counterexamples).

We first solicit a new candidate partial invariant $\varphi$ from the synthesis engine (line 4) given the counterexamples found so far and the grammar $\mathcal{G}$. Then, we check if there are any positive counterexamples that eliminate $\varphi$ (line 5). If one exists, we add it to the set of counterexample and continue searching for more candidates on line 3. If no positive counterexamples can be found, then the partial invariant satisfies the first condition in Definition 1.

We then repeat this for the second condition of Definition 1, seeking implication counterexamples that show that $\varphi$ is not inductive given the previously discovered invariants *Inv* (line 9). If one exists, we add it to the set of counterexamples and return to the loop head on line 3. If there are no implication counterexamples, then we have synthesized a partial invariant that is inductive in

**FO-ICE Algorithm**
INPUT: Protocol description (Initial state formula $Init$, Transition formulas $Trans_a$ for every action $a \in \mathcal{A}$, Safety property $Safe$), Grammar $\mathcal{G}$.
OUTPUT: Sequence of inductive partial invariants $Inv$ such that $\bigwedge Inv$ is an invariant that proves safety (see Section 1.3).
IMPORTS: Model-based synthesis algorithm $ModelSynth$.

(1) $Inv = ()$ // Empty sequence
(2) $Cex = \{\}$
(3) WHILE $True$
(4) $\quad \varphi := ModelSynth(Cex, \mathcal{G})$
(5) $\quad PosCex :=$ model of state $S$ such that $Init(S) \wedge \neg\varphi(S)$
(6) $\quad$ IF $PosCex$ exists
(7) $\quad\quad Cex := Cex \cup \{PosCex\}$
(8) $\quad\quad$ CONTINUE
(9) $\quad ImplCex :=$ model of states $S_1, S_2$ such that $\bigwedge Inv(S) \wedge \varphi(S_1) \wedge Trans_a(S_1, S_2) \wedge \neg\varphi(S_2)$ for some $a \in \mathcal{A}$
(10) $\quad$ IF $ImplCex$ exists
(11) $\quad\quad Cex := Cex \cup \{ImplCex\}$
(12) $\quad\quad$ CONTINUE
(13) $\quad$ ELSE // Inductive partial invariant found
(14) $\quad\quad Inv := Inv \circ \varphi$ // Sequence extension
(15) $\quad\quad Cex := \{C \mid C$ is not an implication counterexample$\}$
(16) $\quad NegCex :=$ model of state $S$ such that $\bigwedge Inv(S) \wedge \neg Safe(S)$
(17) $\quad$ IF $NegCex$ exists
(18) $\quad\quad Cex := Cex \cup \{NegCex\}$
(19) $\quad\quad$ CONTINUE
(20) $\quad$ ELSE
(21) $\quad\quad$ RETURN $Inv$

Figure 3: FO-ICE Algorithm for Synthesizing Invariants

sequence. Therefore, we extend the sequence *Inv* with $\varphi$. We also drop all implication counterexamples found so far (line 15) since they are obviated by $\varphi$. In particular, $\varphi$ eliminates any spurious implication counterexamples (pairs of states that are related by a transition but are not reachable from the initial state).

Finally, after discovering a new inductive invariant, we once again attempt to prove the safety of the protocol by seeking a negative counterexample (line 16). If one exists, we add it to our set *Cex* and continue to line 3. If no negative counterexamples exist, we have in fact proved the protocol safe, so we exit and return *Inv*.

## 2.2 Model-based Formula Synthesis

In this section we describe the engine *ModelSynth* to synthesize formulas from counterexample models. Recall that we only synthesize universally quantified formulas.

Our signatures are such that we can always obtain finite counterexamples. Therefore, we use the encoding technique and synthesis engine *Minisy* from the work in Murali et al. [2020], performing constraint-based synthesis using an SMT solver. We synthesize formulas in prenex form, formulating the grammar $\mathcal{G}$ to only specify the matrix of candidate invariants. Given a finite first-order model $C$, we encode the constraint that a universally quantified formula $\forall \overline{x}.\, \psi$ holds (resp., does not hold) on the model as a ground constraint by unfolding the universal quantifier into a conjunction over the universe of $C$. *Minisy* accepts as input such ground constraints and explores the grammar $\mathcal{G}$ in increasing formula size, returning a single candidate invariant. Since the grammar must fix the number and sort of quantifiers for the matrix, we add an outer loop that enumerates these choices, making one call to *Minisy* for each such choice.

## 2.3 Issues and Mitigation

Having described the basic algorithm we used, we now describe the issues we faced with the algorithm described so far, and the mitigation we came up with.

**Iterative Deepening.**

**Issue**: By default, *Minisy* searches for expressions of progressively larger depth, starting from depth 1 up to a maximum specified depth. However, the number of candidate invariants increases exponentially with this depth. As a result, even with a query depth of size 6, the synthesis query takes 10s of seconds even when there is a single counter-example. Given that the quantified variables of the invariant are chosen in an outer loop, we might be spending a lot of time trying to find an invariant for a bad set of quantified variables, even though for another set of quantified variables, we might find an inductive invariant at a shallow depth.

**Mitigation**: To avoid this issue, we iteratively increase the depth parameter in *Minisy*, and we do so only after we have exhausted all possible set of quantified variables (with a reasonable threshold on the number of quantified variables) up to a given depth.

**Large synthesis queries.**

**Issue:** Early in the project we found that the synthesis queries soon became very slow. For instance, while modeling the "Distributed Lock" protocol, the synthesis queries started by taking 100s of milliseconds, but would later increase to 1 to 2 minutes. This 1200× slowdown happened because the synthesis constraints became quite large as the number of counter-examples increased.

Note that each counter example in the synthesis query consists of two parts – (a) model description and (b) the invariant expression. For example, say we're talking about a positive counter-example $P$ defined by $Init(S) \not\Rightarrow \varphi(S)$. To pass this counter-example to the synthesis engine, we must describe both $S$ and $\varphi(S)$. For the SDL protocol, describing $S$ entails defining the values of $message(src, dst)$ and $lock(N)$ for every value of $src$, $dst$ and $N$. Conceptually, this can be thought of as adding rows in a table – one row for every valuation of $(src, dst)$ pair for example. Concretely, we define these relations using *ite* expressions in Z3. As counter-example models become larger, we found that Z3 spent majority of the query time in just *simplifying* these large *ite* expressions.

Similarly, the invariant expression also becomes larger with larger models. This is true because we unfold the universal quantifiers of this expression.

**Mitigation:** We mitigate these issues using two methods:

(1) We compress the model description: Instead of using a conceptual 'table' to model a state relation, we model it as a decision tree. We use the ID3 algorithm to obtain an efficient representation of the relation. This reduces the depth of the *ite* expressions considerably. The response time for a query of similar complexity reduced significantly.

(2) Unfold quantifiers lazily: Instead of expanding the universal quantifiers in the invariant over the entire universe eagerly, we do the unfolding over different rounds of synthesis queries. The intuition behind this is that the solver might be able to come up with reasonable candidates with fewer constraints. We call this optimization "Lazy expansion".

**Trivial candidate invariants.**

**Issue:** One of the first issues we faced was that the generated invariants were often trivially true, or trivially equivalent to previously known constraints.

**Mitigation:**

First we mitigated this issue by adding a rather ad-hoc constraint to the synthesis query – that the generated invariant is False for *some* valuation of its quantified variables. This gave invariants that weren't trivially true, however it kept giving invariants that were essentially re-writes of previously discovered invariants. Another ad-hoc constraint was added which asserted that the previously known invariants do not imply the candidate invariant. However, this resulted in quantified expressions.

Finally, a more principled approach was adopted, which we call the **witness idea**. Essentially, we assert that there exists some model, where the previously known invariants hold, but the candidate invariant doesn't hold. In other words, we ask the synthesis engine to not only give us a candidate invariant, but also give us a proof/witness in the form of a model that previously known invariants were unable to eliminate, but the new invariant eliminates.

We experimented with two variants of the witness idea – one that required that the eliminated state was an unsafe state, that is, we asked for a negative-counter example; and the other variant did not apply that restriction. The former imposes a stronger notion of progress – because every new candidate invariant avoids at least some bad state. However, it is also a stronger ask, making the synthesis query harder to solve. We eventually settled for the simpler option.

# 3 Results

We evaluate our approach on two protocols: Ricart-Agrawala and Distributed Lock. Both protocols are part of the benchmark used by DistAI Yao et al. [2021]. DistAI proves both protocols to be safe, whereas our method is only able to prove safety of the Ricart-Agrawala protocol. For the Distributed

| Optimizations | | | Results | |
|---|---|---|---|---|
| IterDeep | ID3 | Lazy Expansion | #invariants found | #synthesis queries |
| - | - | - | 2 | 12 |
| - | - | ✔ | 2 | 16 |
| - | ✔ | - | 6 | 35 |
| - | ✔ | ✔ | 3 | 31 |
| ✔ | - | - | 15 | 779 |
| ✔ | - | ✔ | 12 | 1016 |
| ✔ | ✔ | - | 19 | 979 |
| ✔ | ✔ | ✔ | 10 | 955 |

Table 1: **Effect of optimizations on the Distributed Lock protocol**: IterDeep and the ID3 optimizations together found the most number of invariants. Lazy Expansion did not help and instead often hurt the performance. None of the optimizations were able to fully prove safety of the Distributed Lock protocol.

Lock protocol, our method can come up with several useful inductive invariants, however they are not sufficient to prove the safety property. It is worth noting that even the approach adopted by SWISS Hance et al. [2021] is unable to fully prove this protocol safe.

We now discuss the effect of various optimizations we came up with in order to improve our method's performance on the Distributed Lock protocol. We do not discuss the effect of these optimizations for the Ricart-Agrawala protocol because they don't make much difference given that the original method itself is able to prove the safety fairly quickly (in less than a minute).

Table 1 summarizes our experiments for the Distributed Lock protocol. The experiments were performed with the optimizations we proposed in the earlier section. Note that all variants were run with the Witness Optimization because without that the system was generating largely trivial invariants. Also note that the experiments without the IterDeep optimization were stopped at 30 minutes because they weren't making meaningful progress. Experiments with the IterDeep optimization were run for 90 minutes each.

As mentioned in the table, none of the optimizations helped prove safety of the distributed lock protocol. We investigated why that was the case by comparing the invariants generated by DistAI. All DistAI invariants only had 3 terms/quantified variables, which was well within our search space. The reason we were not able to prove the safety of this protocol was because of dependency between the invariants. Our approach discovers invariants one at a time, building up on previously discovered invariants. DistAI completely sidesteps this issue by instead discovering conjunctions of invariants. As a result, they do not face the dependency issue at all.

Our code can be found at https://github.com/thakkarparth007/cs598mp-project.

# 4 Conclusions and Future Work

In this work we have explored the effectiveness of FO-ICE in synthesizing invariants for distributed protocols. Despite somewhat unsatisfactory results compared to prior work, there are some promising future directions. First, the approach in the work in Yao et al. [2021] is essentially that of the Houdini algorithm for learning conjunctive invariants Flanagan and Leino [2001]. In particular, the predicates are quantified FO sentences. They key idea that makes Yao et al. [2021] work is their clever use of implication ordering during enumeration, that allows the Houdini algorithm to scale to this problem.

However, the method in Yao et al. [2021] does not scale when invariants are not purely universal, which suggests that the general problem may be effectively tackled by approaches like Sorcar Neider et al. [2019] that fare better than Houdini when the total number of predicates is large but the number of predicates appearing in the invariant is not.

More relevant to the case of universal invariants is the direction of moving towards handling distributed protocols whose verification conditions do not fall into the EPR fragment Grädel [1989], Piskac et al. [2010], Padon et al. [2017]. This appears to be a restriction that is talked about as being important in the works in Yao et al. [2021], Hance et al. [2021], but handling a more general FO fragment using quantifier instantiation techniques Löding et al. [2018] is an interesting direction. We expect that doing this in our algorithm instead of employing a decision procedure will result in cases where a valid formula may not be provable. As such, our model may not be counterexamples to validity, but only counterexamples to non-provability. The idea of using non-provability counterexamples has been explored in the work in Murali et al. [2020] where it is shown that such examples do provide a good signal to guide synthesis towards useful formulas. This leads us to believe that the restriction to the EPR fragment (or decidable fragments in general) may not be necessary.

Lastly, on the front of techniques we have found that constraint-based synthesis (using existing solvers) does not scale for synthesizing formulas using counterexample models. Enumerative synthesis fares somewhat better for model-guided synthesis in certain scenarios as shown in the work in Hance et al. [2021], but cannot be expected to succeed in scenarios where invariants are large formulas deep in the grammar. This showcases the need for pursuing algorithms specific to model-guided synthesis of formulas. We have explored some initial ideas for this in our work with the optimizations discussed in Section 2.3.

# References

Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. *Dependable Software Systems Engineering*, 2015.

Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, page 500–517, Berlin, Heidelberg, 2001. Springer-Verlag. ISBN 3540417915.

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. ISBN 978-3-319-08867-9.

Erich Grädel. Complexity of formula classes in first order logic with functions. In J. Csirik, J. Demetrovics, and F. Gécseg, editors, *Fundamentals of Computation Theory*, pages 224–233, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg. ISBN 978-3-540-48180-5.

Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. Finding invariants of distributed systems: It's a small (enough) world after all. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 115–131. USENIX Association, April 2021. ISBN 978-1-939133-21-2. URL https://www.usenix.org/conference/nsdi21/presentation/hance.

Jason R. Koenig, Oded Padon, Sharon Shoham, and Alex Aiken. Inferring invariants with quantifier alternations: Taming the search space explosion. In *Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part I*, page 338–356, Berlin, Heidelberg, 2022. Springer-Verlag. ISBN 978-3-030-99523-2. doi: 10.1007/978-3-030-99524-9_18. URL https://doi.org/10.1007/978-3-030-99524-9_18.

Christof Löding, P. Madhusudan, and Lucas Peña. Foundations for natural proofs and quantifier instantiation. *PACMPL*, 2(POPL):10:1–10:30, 2018. doi: 10.1145/3158098.

Adithya Murali, Lucas Peña, Christof Löding, and P. Madhusudan. Synthesizing lemmas for inductive reasoning. *CoRR*, abs/2009.10207, 2020. URL https://arxiv.org/abs/2009.10207.

Daniel Neider, Shambwaditya Saha, Pranav Garg, and P. Madhusudan. Sorcar: Property-driven algorithms for learning conjunctive invariants. In Bor-Yuh Evan Chang, editor, *Static Analysis*, pages 323–346, Cham, 2019. Springer International Publishing. ISBN 978-3-030-32304-2.

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, page 614–630, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342612. doi: 10.1145/2908080.2908118. URL https://doi.org/10.1145/2908080.2908118.

Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made epr: Decidable reasoning about distributed protocols. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017. doi: 10.1145/3140568. URL https://doi.org/10.1145/3140568.

Ruzica Piskac, Leonardo de Moura, and Nikolaj Bjørner. Deciding effectively propositional logic using dpll and substitution sets. *Journal of Automated Reasoning*, 44(4):401–424, Apr 2010. ISSN 1573-0670. doi: 10.1007/s10817-009-9161-6. URL https://doi.org/10.1007/s10817-009-9161-6.

Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. DistAI: Data-Driven automated invariant learning for distributed protocols. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 405–421. USENIX Association, July 2021. ISBN 978-1-939133-22-9. URL https://www.usenix.org/conference/osdi21/presentation/yao.