Synthesis of Distributed Agreement-Based Systems with Efficiently-Decidable Verification

Nouraldin Jaber[1], Christopher Wagner[1], Swen Jacobs[2], Milind Kulkarni[1], and Roopsha Samanta[1]

[1] Purdue University, West Lafayette, USA
{njaber,wagne279,milind,roopsha}@purdue.edu
[2] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
jacobs@cispa.de

Abstract. *Distributed agreement-based* (DAB) systems use common distributed agreement protocols such as leader election and consensus as building blocks for their target functionality. While automated verification for DAB systems is undecidable in general, recent work identifies a large class of DAB systems for which verification is *efficiently-decidable*. Unfortunately, the conditions characterizing such a class can be opaque and non-intuitive, and can pose a significant challenge to system designers trying to model their systems in this class.

In this paper, we present a synthesis-driven tool, Cinnabar, to help system designers building DAB systems ensure that their intended designs belong to an efficiently-decidable class. In particular, starting from an initial *sketch* provided by the designer, Cinnabar generates sketch completions using a counterexample-guided procedure. The core technique relies on compactly encoding root-causes of counterexamples to varied properties such as efficient-decidability and safety. We demonstrate Cinnabar's effectiveness by successfully and efficiently synthesizing completions for a variety of interesting DAB systems including a distributed key-value store and a distributed consortium system.

1 Introduction

Distributed system designers are increasingly embracing the incorporation of formal verification techniques into their development pipelines [8,11,14,33]. The formal methods community has been enthusiastically responding to this trend with a wide array of modeling and verification frameworks for prevalent distributed systems [31,19,17,34]. A desirable workflow for a system designer using one of these frameworks is to (1) provide a framework-specific model and specification of their system, and (2) *automatically* verify if the system model meets its specification.

However, the problem of algorithmically checking if a distributed system is correct for an *arbitrary* number of processes, i.e., the *automated parameterized verification problem*, is undecidable, even for finite-state processes [5,36]. To circumvent undecidability, the system designer must be involved, *one way* *or another*, in the verification process. *Either* the designer may choose a semi-automated verification approach and use their expertise to "assist" the verifier by providing inductive invariants [34,27,17,38]. *Or,* the designer may choose a fully-automated verification approach that is only applicable to a restricted class of system models [18,19,26,7] and use their expertise to ensure that the model of their system belongs to the decidable class. This begs the question—*for each workflow, how can we further simplify the system designer's task?* While effective frameworks have been developed to aid the designer in discovering inductive invariants for the first workflow (e.g., Ivy [31], I4 [28]), there has been little emphasis on aiding the designer to build *decidability-compliant* models of their systems for the second workflow.

In this paper, we present a synthesis-driven approach to help system designers using the second workflow to build models that are *both* decidability-compliant and correct. Thus, our approach helps designers to construct models that belong to a decidable class for automated, parameterized verification, and can be automatically verified to be safe for any number of processes.

In particular, we instantiate this approach in a tool, Cinnabar, that targets an existing framework, QuickSilver, for modeling and automated verification of *distributed agreement-based (DAB) systems* [19]. Such systems use *agreement protocols* such as leader election and consensus as building blocks. QuickSilver enables modular verification of DAB systems by providing a modeling language, Mercury, that allows designers to model *verified* agreement protocols using inbuilt language primitives, and identifying a class of Mercury models for which the parameterized verification problem is *efficiently decidable.*

Unfortunately, this *efficiently-decidable* class of Mercury models is characterized using conditions that are rather opaque and non-intuitive, and can pose a significant challenge to system designers trying to model their systems in this class. The designer is responsible for understanding the conditions, and manually modifying their system model to ensure it belongs to the efficiently-decidable class of Mercury. This process can be both tedious and error-prone, even for experienced system designers.

Cinnabar demonstrates that synthesis can be used to automatically build models of DAB systems that belong to the efficiently-decidable fragment of Mercury and are correct.

Contributions. The key contributions of this paper are:

1. *A synthesis-driven method for building efficiently-decidable, correct* Mercury *models* (Sec. 3). Starting from an initial *sketch* of the system design provided by the designer, Cinnabar generates a *sketch completion* that (i) belongs to the efficiently-decidable class of Mercury *and* (ii) is correct.
2. *A counterexample-guided synthesis procedure that leverages an efficient, extensible, multi-stage architecture* (Sec. 4). We present a procedure that involves a learner that proposes completions of the Mercury sketch, and a teacher that checks if the completed model belongs to the efficiently-decidable class of Mercury and is correct. To enable efficient synthesis using this procedure, we propose an architecture that proceeds in stages.

The initial stages focus on checking if a completed model is in the efficiently-decidable class while the latter stages focus on checking if a completed model is also correct. To enable efficiency, when a candidate completion fails at any stage, the architecture helps the learner avoid " similar" completions by *extracting* a *root-cause* of the failure and *encoding* the root-cause as an additional constraint for the learner. Each stage is equipped with a counterexample extraction strategy tailored to the *property* checked in that stage. The encoding procedure, on the other hand, is property-agnostic—it is able to encode the root-cause of any failure regardless of the stage that extracts it. The separation of the counterexample extractions and the encoding allows the architecture to be extensible—one can add a new stage with a new counterexample extraction strategy, and leverage the existing encoding.

3. *The* CINNABAR *tool* (Sec. 5). We develop a tool, CINNABAR, to help system designers build MERCURY models of DAB systems. CINNABAR employs QUICKSILVER as its teacher and the Z3 SMT solver as its learner. CINNABAR is able to successfully and efficiently complete MERCURY sketches of various interesting distributed agreement-based systems.

2 The MERCURY Parameterized Synthesis Problem

We first briefly review the syntax and semantics of MERCURY [19], a modeling language for distributed systems that build on top of verified agreement protocols such as leader election and consensus. Then, we formalize the synthesis problem.

2.1 Review: MERCURY Systems

```
process DistributedStore
variables
  int[1,5] cmd
events
  env rz doCmd : int[1,5]
initial location Candidate
  on partition<elect>(All, 1)
    win: goto Leader
    lose: goto Replica
location Leader
  on recv(doCmd) do
    cmd := doCmd.payld
    if(cmd = 3) goto Return
    else goto RepCmd
...
```

MERCURY *Process Definition.* A MERCURY system is composed of an arbitrary number of n identical MERCURY system processes with process identifiers $1, \ldots, n$ and one environment process. The programmer specifies a system process definition P that consists of (i) a set V of *local variables* with finite domains, (ii) a set E of *events* used to communicate between processes, and (iii) a set of *locations* that the processes can move between. Each event e in E incarnates an *acting action* $A(e)$ and a *reacting action* $R(e)$ (e.g., for a rendezvous event, the acting (resp. reacting) action is the send (resp. receive) of that event). All processes start in a location denoted `initial`. Each location contains a set of *action handlers* a process in that location can execute. Each handler has an associated action, a Boolean guard over the local variables, and a set of update statements. A partial process definition is depicted on the right.

The language supports five different types of events, namely, broadcast, rendezvous, partition, consensus, and internal. The *synchronous* broadcast (resp.

rendezvous) *communication* event type is denoted `br` (resp. `rz`) and indicates an event where one process synchronously communicates with all other processes (resp. another process). The *agreement* event type partition, denoted `partition`, indicates an event where a set of processes agree to partition themselves into winners and losers. For instance, in the figure, `partition<elect>(All,1)` denotes a leader election round with identifier `elect` where `All` processes elect `1` winning process that moves to the `Leader` location, while all other losing processes move to the `Replica` location. The *agreement* event type consensus, denoted `consensus`, indicates an event where a set of processes, each proposing one value, reach consensus on a given set of decided values. For instance, `consensus<vcCmd>(All,1,cmd)` denotes a consensus round with identifier `vcCmd` where `All` processes want to agree on `1` decided value from the set of proposed values in the local variable `cmd`. Finally, the internal event indicates an event where a process is performing its own internal computations. For a communication event, the acting action is a send, while the reacting action is a receive. For a partition event, the acting action is a win, while the reacting action is a lose. Finally, for a consensus event, the acting action is proposing a winning value, while the reacting action is proposing a losing value. We denote by $A(E)$ and $R(E)$ the set of all acting and reacting actions, respectively.

The updates in an action handler may contain send, assignment, goto, and/or conditional statements. Assignment statements are of the form `lhs` $\coloneqq$ `rhs` where `lhs` is a local variable and `rhs` is an expression of the appropriate type. The goto statement `goto` ℓ causes the process to switch to location ℓ (i.e., it can be thought of as the assignment statement $v_{loc} \coloneqq \ell$, where v_{loc} is a special "location variable" that stores the current location of the process). The conditional statements are of the expected form: `if(cond) then...else...`. We denote by H the set of all handlers in the process, and for each handler $h \in H$ we denote its action, guard, and updates as $a(h)$, $g(h)$, and $u(h)$, respectively.

Local Semantics. The local semantics $[\![P]\!]$ of a process P is expressed as a state-transition system (S, s_0, E, T), where S is the set of local states, s_0 is the initial state, E is the set of events, and $T \subseteq S \times \{A(E) \cup R(E)\} \times S$ is the set of transitions of $[\![P]\!]$. A state $s \in S$ is a valuation of the variables in V. We let $s(v)$ denote the value of the variable v in state s.

The set of action handlers associated with all acting and reacting actions of all events induces the transitions in T. In particular, a transition $t = s \xrightarrow{a} s'$ based on action handler h over action a is in T iff the guard $g(h)$ evaluates to *true* in s and s' is obtained by applying the updates $u(h)$ to s.

Global Semantics. The global semantics $[\![P, n]\!]$ of a Mercury system $P_1 || \ldots || P_n || P_e$ consisting of n identical processes $P_1, \ldots, P_n$ and an environment process P_e (with local state space S_e) is expressed as a transition system (Q, q_0, E, R), where $Q = S^n \times S_e$ is the set of global states, q_0 is the initial global state, E is the set of events, and $R \subseteq Q \times E \times Q$ is the set of global transitions of $[\![P, n]\!]$.

The set of events E induce the transitions in R. As is the case for events, there are five types of global transitions: broadcast, rendezvous, partition, consensus, and internal. In particular, a transition $r = q \xrightarrow{e} q'$ for some broadcast event e

is in R iff the send local transition $q[i] \xrightarrow{A(e)} q[i]'$ is in T for some process P_i, and the receive local transition $q[j] \xrightarrow{R(e)} q[j]'$ is in T for every other process P_j with $j \neq i$. The remaining global transitions can be formalized similarly.

A trace of a MERCURY system is a sequence $q_0, q_1, \ldots$ of global states such that for every $i \geq 0$, the global transition $q_i \xrightarrow{e} q_{i+1}$ for some event e is in R. A global state q is *reachable* if there is a trace that ends in it.

Permissible Safety Specifications. QUICKSILVER targets parameterized verification for a class of properties called *permissible* safety specifications that disallow global states where m or more processes, for some fixed number m, are in some subset of the local states. We denote by $\phi_s(n)$ the permissible safety specifications provided by the designer for a system with n processes. A MERCURY system is safe if there are no reachable error states in its global semantics. We denote that as $[\![P, n]\!] \models \phi_s(n)$.

The Efficiently-Decidable Fragment. QUICKSILVER identifies a fragment of MERCURY for which the parameterized verification problem of a large class of safety properties is *efficiently-decidable*. In particular, a pair $\langle P, \phi\rangle$ of a MERCURY process P and a safety specification ϕ is in the efficiently-decidable fragment of MERCURY if it satisfies *phase-compatibility* and *cutoff-amenability* conditions. For such a pair, a *cutoff* number c of processes can be computed and the parameterized verification problem can be reduced to the verification of the cutoff-sized system (i.e., $\forall n : [\![P, n]\!] \models \phi_s(n) \Leftrightarrow [\![P, c]\!] \models \phi_s(c)$).

During verification, QUICKSILVER computes a set of *phases* that an execution of the system goes through. On a high level, the phase-compatibility conditions ensure that the system moves between phases through "globally-synchronizing" events (i.e., broadcast, partition, or consensus), and that all processes in the same phase can participate in further globally-synchronizing events. This ensures that the system's ability to move between phases is *independent* of the number of processes in the system. The cutoff-amenability conditions ensure that an error state, where m processes are in a subset of the local states violating some safety specification, is reachable in a system of any size iff it is reachable in a system with exactly m processes. If any of these conditions fails, the designer must modify the process definition manually and attempt the verification again. We denote by $[\![P]\!] \models \phi_{pc}$ (resp. $[\![P]\!] \models \phi_{ca}$) that the MERCURY process P with local semantics $[\![P]\!]$ satisfies phase-compatibility (resp. cutoff-amenability) conditions.

2.2 MERCURY Process Sketch

Let us extend MERCURY's syntax to allow process *sketches* that can be completed by a synthesizer. In particular, we allow the process definition to include a set of *uninterpreted functions* that can replace various expressions in MERCURY such as the Boolean expression `cond` in the `if(cond) then ... else ...`, the target locations of `goto` statements, and the `rhs` of assignments. [3] As is standard, each uninterpreted function f is equipped with a signature determining its

[3] Such uninterpreted functions are sufficient to be a building block for more complex expressions and statements (See, for instance, the SKETCH Language [35]).

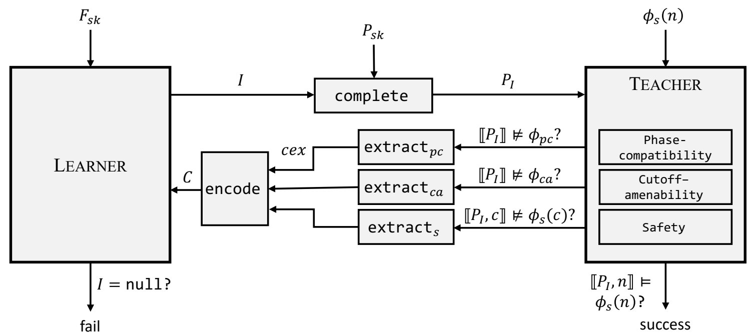

Fig. 1: Overview of CINNABAR's architecture.

list of named, typed parameters and its return type. A valid list of arguments `arg` for some function f is a list of values with types that match the function's parameter list. Applying a function f to a valid list of arguments `arg` is denoted by $f($`arg`$)$. Additionally, we define a *function interpretation* $I(f)$ of an uninterpreted function f as a mapping from every valid list of arguments of f to a valid return value.

A MERCURY process definition P that contains one or more uninterpreted functions is called a *sketch*, and is denoted P_{sk}. We denote by F_{sk} the set of all uninterpreted functions in a sketch P_{sk}. An interpretation I of the set F_{sk} of uninterpreted functions is then a mapping from every uninterpreted function $f_{sk} \in F_{sk}$ to some function interpretation $I(f_{sk})$.

For some process sketch P_{sk} and some interpretation I of the set F_{sk} of uninterpreted functions in P_{sk}, we denote by P_I the *interpreted process sketch* obtained by replacing every uninterpreted function $f_{sk} \in F_{sk}$ in the sketch P_{sk} with its function interpretation $I(f_{sk})$ according to the interpretation I.

2.3 Problem Definition

We now define the parameterized synthesis problem for MERCURY systems.

Definition 1 (MERCURY Parameterized Synthesis Problem (MPSP)). *Given a process sketch P_{sk} with a set of uninterpreted functions F_{sk}, an environment process P_e, and permissible safety specification $\phi_s(n)$, find an interpretation I of uninterpreted functions in F_{sk} such that the system $P_{I,1}||\ldots||P_{I,n}||P_e$ is safe for any number of processes, i.e., $\forall n : [\![P_I, n]\!] \models \phi_s(n)$.*

3 Constraint-Based Synthesis for MERCURY Systems

Architecture. To solve MPSP, we propose a multi-stage, counterexample-based architecture, shown in Fig. 1, with the following components:

- LEARNER: a constraint-solver that accepts a set C of *constraints* over the uninterpreted functions F_{sk} and generates interpretations I satisfying these constraints (i.e., $I \models C$). Specifically, a constraint $\mathtt{c} \in C$ is a well-typed Boolean formula over uninterpreted function applications.
- TEACHER: a component capable of checking phase-compatibility, cutoff-amenability, safety, and *liveness*[4] of MERCURY systems. We refer to these four conditions as *properties*.
- `complete`: a component that builds an interpreted process sketch P_I from a process sketch P_{sk} and an interpretation I provided by the learner.
- $\mathtt{extract}_{prop}$: a *property-specific* component to extract a counterexample cex, capturing the root cause of a violation, if the TEACHER determines that a property *prop* from the above-mentioned properties is violated.
- `encode`: a novel *property-agnostic* component that encodes counterexamples generated by `extract` components into additional constraints for the learner.

Synthesis Procedure. CINNABAR instantiates this architecture as shown in Algo. 1. The algorithm starts with an empty set of constraints, C (Line 2) over the set F_{sk} of uninterpreted functions in the process sketch P_{sk}. In each iteration, it checks if there exists an interpretation I of the uninterpreted functions that satisfies all the constraints collected so far (Line 4). If such an interpretation is found, it is used to obtain an interpreted process sketch P_I (Line 6). Then, the algorithm checks if the system described by P_I is phase-compatible and cutoff-amenable. If so, a cutoff c is computed (Line 13) and the c-sized system is checked to be safe. The cutoff-amenability stage is similar to phase-compatibility and is hence omitted from the algorithm. At any stage, if the process fails to satisfy any of these properties (e.g., a counterexample cex_p to phase-compatibility is found on Line 8), the root-cause of the failure is extracted and encoded into a constraint for the learner to rule out the failure (e.g., Line 10).

Algorithm 1: Solving MPSP.

1 **procedure** $\mathtt{Synth}(P_{sk}, \phi_s(n), \phi_l(c))$
2    $C = \varnothing$
3   **while** *true* **do**
4      $I = \mathtt{interpret}(F_{sk}, C)$
5   **if** $I \neq \textit{null}$ **then**
6        $P_I = \mathtt{complete}(P_{sk}, I)$
7        $[\![P_I]\!] = \mathtt{buildLS}(P_I)$
8        $cex_p = \mathtt{findPhCoCE}([\![P_I]\!])$
9   **if** $cex_p \neq \textit{null}$ **then**
10          $C = C \cup \neg\, \mathtt{encode}(cex_p)$
11   **Continue**
12   ... ▷ check cutoff-amenability
13        $c = \mathtt{compCutoff}(P_I, \phi_s(n))$
14        $[\![P_I, c]\!] = \mathtt{buildGS}(P_I, c)$
15        $cex_s = \mathtt{findSaCE}([\![P_I, c]\!], \phi_s(c))$
16   **if** $cex_s \neq \textit{null}$ **then**
17          $C = C \cup \neg\, \mathtt{encode}(cex_s)$
18   **Continue**
19   **return** P_I
20   **else**
21   **return** `null`

[4] While MPSP targets permissible safety specifications, in order to improve the *quality* of the interpreted process sketch P_I, we extend MERCURY with *liveness* specifications to help rule out trivial completions that are safe. We emphasize that such specifications are only used as a tool to improve the quality of synthesis, and are only guaranteed for the cutoff-sized system, as opposed to safety properties that are guaranteed for any system size.

Note that these stages are checked sequentially due to the inherent dependency between them: (i) the system can only be cutoff amenable if it is phase compatible, and (ii) one can only check safety after a cutoff has been computed.

Lemma 1. *Assuming that the teacher is sound and the learner is complete for finite sets of interpretations, Algo. 1 for solving MPSP is sound and complete.*

Proof. Soundness follows directly from the soundness of the teacher. Completeness follows from that the encoding and extraction procedures ensure progress by eliminating at least the current interpretation at each iteration, and the finiteness of the set of interpretations. Finiteness follows from (i) the finite number of uninterpreted functions in a sketch P_{sk}, (ii) the finiteness of the domain of each local variable, and (iii) the finiteness of the number of local variables in P_{sk}.

In the remainder of this section, we describe the property-agnostic `encode` component in Algo. 1. In the following section, we describe our implementation of our synthesis procedure specialized to a QuickSilver-based teacher and property-specific extraction procedures.

Property-Agnostic Counterexample Encoding Procedure

We first describe the necessary augmentation of local semantics with *disabled transitions* needed for Cinnabar's counterexample extraction and encoding. While such transitions are not relevant when reasoning about a "concrete" process definition (i.e., one with no uninterpreted functions), they are quite important when extracting an *explanation* for why some conditions (e.g., phase-compatibility) fail to hold on $[\![P]\!]$.

***Augmented Local Semantics of the* Mercury *Process* $\mathbf{P_I}$.** We extend the definition of the local semantics of a Mercury interpreted process sketch P_I to be $[\![P_I]\!] = (S_I, s_0, E, T_I, T_I^{dis})$ where S_I, s_0, E, and T_I are defined as before and T_I^{dis} is the set of *disabled transitions* under the current interpretation I. In particular, a disabled transition $t = s \xrightarrow{a} \perp$ based on action handler h over action a is in T_I^{dis} iff the guard $g(h)$ evaluates to $false$ in s. The symbol $\perp$ here indicates that no local state is reachable, since the guard is disabled.

Additionally, we say a transition $t = s \xrightarrow{a} s'$ based on action handler h over action a is a *sketch transition* if h contains no uninterpreted functions in its guard or updates. A local state $s \in S_I$ is *concrete* if (i) s is the initial state s_0, or (ii) there exists a sketch transition $s' \rightarrow s$ where s' is concrete. In other words, a local state s is concrete if there exists a path from the initial state s_0 to s that is composed purely of sketch transitions and hence is always reachable regardless of the interpretation we obtain from the learner.

We now formalize counterexamples for phase-compatibility and cutoff amenability properties then present an encoding procedure for such counterexamples. The encoding is *exact* in the sense that a generated constraint `c` corresponding to some counterexample *cex* rules out exactly all interpretations I where an interpreted process sketch P_I exhibits *cex* (as opposed to an over-approximation where c would rule out interpreted process sketches that do not exhibit *cex*, or an under-approximation where c would allow interpreted process sketches that do exhibit *cex*). Additionally, the encoding is *property-agnostic* in the sense that it can handle counterexamples for any property failure.

Counterexamples. Recall that a candidate process P_I based on some process sketch P_{sk} and interpretation I has the local semantics $[\![P_I]\!] = (S_I, s_0, E, T_I, T_I^{dis})$. A counterexample *cex* to phase-compatibility (resp. cutoff-amenability) is a "subset" of the local semantics $[\![P_I]\!]$ such that $cex \not\models \phi_{pc}$ (resp. $cex \not\models \phi_{ca}$). We say that *cex* is a subset of $[\![P_I]\!]$, denoted $cex \subseteq [\![P_I]\!]$, when it has a subset of its enabled and disabled transitions, i.e., $cex = (S_I, s_0, E, T'_I \subseteq T_I, T'^{dis}_I \subseteq T_I^{dis})$.

Encoding Counterexamples. Let $\mathcal{C}$ be the set of all well-typed constraints that the learner accepts. The encoding of counterexample $cex = (S_I, s_0, E, T_I, T_I^{dis})$ w.r.t. interpretation I is a formula $\langle\!\langle cex \rangle\!\rangle_I \in \mathcal{C}$ defined as:

$$\langle\!\langle cex \rangle\!\rangle_I = \Big(\bigwedge_{t_{en} \in T_I} \langle\!\langle t_{en} \rangle\!\rangle_I \Big) \wedge \Big(\bigwedge_{t_{dis} \in T_I^{dis}} \langle\!\langle t_{dis} \rangle\!\rangle_I \Big),$$

where $\langle\!\langle t_{en} \rangle\!\rangle_I$ (resp. $\langle\!\langle t_{dis} \rangle\!\rangle_I$) is an encoding of an enabled (resp. disabled) local transition. Note that $\langle\!\langle cex \rangle\!\rangle_I$ is satisfied under interpretation I (i.e., $I \models \langle\!\langle cex \rangle\!\rangle_I$) and implies that $cex \subseteq [\![P]\!]$. An encoding of some enabled transition $t_{en} = s \xrightarrow{a} s'$ based on action handler h over action a is defined as:

$$\langle\!\langle s \xrightarrow{a} s' \rangle\!\rangle_I = \langle\!\langle s \rangle\!\rangle_I \wedge \langle\!\langle a : s \rangle\!\rangle_I \wedge \langle\!\langle s' : s, a \rangle\!\rangle_I,$$

where:

1. the predicate $\langle\!\langle s \rangle\!\rangle_I$ indicating that the source state s is reachable from the initial state s_0 under interpretation I. If s is concrete, $\langle\!\langle s \rangle\!\rangle_I$ is simply *true* (i.e., s is always reachable regardless of I). Otherwise, $\langle\!\langle s \rangle\!\rangle_I$ is defined as follows. Let $\mathcal{P}$ be the set of all paths from the initial state s_0 to state s. Then, $\langle\!\langle s \rangle\!\rangle_I \coloneqq \bigvee_{p \in \mathcal{P}} \langle\!\langle p \rangle\!\rangle_I$, where $\langle\!\langle p \rangle\!\rangle_I$ for some path p consisting of local transitions $t_1, \ldots, t_i$ is defined as $\langle\!\langle t_1 \rangle\!\rangle_I \wedge \ldots \wedge \langle\!\langle t_i \rangle\!\rangle_I$.
2. the predicate $\langle\!\langle a : s \rangle\!\rangle_I$ indicating that the process can perform action a from state s. The predicate $\langle\!\langle a : s \rangle\!\rangle_I$ is defined as follows: $\langle\!\langle a : s \rangle\!\rangle_I \coloneqq (g(h)[s(V)/V] = true)$, where $g(h)[s(V)/V]$ is the guard $g(h)$ with each local variable $v \in V$ replaced by its value $s(v)$ in state s.

   Example. Let $\mathtt{uf}(x, y)$ be an uninterpreted function over local `int` variables x and y. Let the local state $s \coloneqq \{v_{loc} = \mathtt{F}, x = 1, y = 2\}$, and let the local guard of action handler h over action a in location F be $g \coloneqq \mathtt{uf}(x, y) > 7 \vee x = 2$. Then $\langle\!\langle a : s \rangle\!\rangle_I = \big((\mathtt{uf}(s(x), s(y)) > 7 \vee s(x) = 2) = true) \big)$ which is $\big((\mathtt{uf}(1, 2) > 7 \vee 1 = 2) = true \big)$ which simplifies to $\mathtt{uf}(1, 2) > 7$.
3. the predicate $\langle\!\langle s' : s, a \rangle\!\rangle_I$ indicating that s goes to s' on action a. The predicate $\langle\!\langle s' : s, a \rangle\!\rangle_I$ is defined as follows. Let $u(h)$ denote the set of updates of the form `lhs` ≔ `rhs` of handler h over action a. Then, $\langle\!\langle s' : s, a \rangle\!\rangle_I \coloneqq \bigwedge_{\mathtt{lhs} := \mathtt{rhs} \in u(h)} s'(\mathtt{lhs}) = \mathtt{rhs}[s(V)/V]$.

   Example. Let the set of updates have the single update $x \coloneqq \mathtt{uf}(y, z)$ and s, s' be $\{v_{loc} = \mathtt{F}, x = 1, y = 2, z = 3\}$ and $\{v_{loc} = \mathtt{D}, x = 5, y = 2, z = 3\}$. Then $\langle\!\langle s' : s, a \rangle\!\rangle_I$ is: $s'(x) = \mathtt{uf}(s(y), s(z))$ which is $\mathtt{uf}(2, 3) = 5$.

An encoding of some disabled transition $t_{dis} = s \xrightarrow{a} \bot$ in cex is defined as $\langle\!\langle t_{dis} \rangle\!\rangle_I = \langle\!\langle s \rangle\!\rangle_I \land \langle\!\langle \neg a : s \rangle\!\rangle_I$ where $\langle\!\langle s \rangle\!\rangle_I$ is as before and the predicate $\langle\!\langle \neg a : s \rangle\!\rangle_I$, indicating that the process cannot perform action a from state s, is defined as follows: $\langle\!\langle \neg a : s \rangle\!\rangle_I \coloneqq (g(h)[s(V)/V] = false)$.

The intuition behind breaking a transition's encoding to various predicates is that some phase-compatibility conditions leave parts of a transition unspecified. For instance, the predicate "the local state s can react to event e" corresponds to a local transition $s \xrightarrow{R(e)} * \in T_I$ with encoding $\langle\!\langle s \rangle\!\rangle_I \land \langle\!\langle R(e) : s \rangle\!\rangle_I$.

Finally, to rule out any interpretation I that exhibits cex, we add the constraint $\mathtt{c} = \neg\langle\!\langle cex \rangle\!\rangle_I$ to the learner.

Encoding Counterexamples to Safety Properties. Similar to the local semantics, we extend the definition of the global semantics $[\![P_I, n]\!]$ of a MERCURY system $P_{I,1}||\ldots||P_{I,n}||P_e$ to be $[\![P_I, n]\!] = (Q_I, q_0, E, R_I, R_I^{dis})$, where Q_I, q_0, E, and R_I are defined as before and R_I^{dis} is the set of *disabled global transitions* under the current interpretation I. Then, a counterexample cex to safety is a "subset" of the *global* semantics $[\![P_I, c]\!]$ such that $cex \not\models \phi_s(c)$. Encoding of such a counterexample cex is formalized as before, with the encoding of an enabled global transition r in cex being a formula $\langle\!\langle cex \rangle\!\rangle_I \in \mathcal{C}$ computed as follows. For some global transition $r = q \xrightarrow{e} q'$, we denote by $active(r)$ the local transitions that processes in q locally use to end in q'. That is, $active(r) = \{t \in T_I \mid \exists P_{I,i} : t = q[i] \xrightarrow{A(e)} q'[i] \lor t = q[i] \xrightarrow{R(e)} q'[i]\}$ We then define the encoding $\langle\!\langle r \rangle\!\rangle_I$ as: $\langle\!\langle r \rangle\!\rangle_I = \bigwedge_{t \in active(r)} \langle\!\langle t \rangle\!\rangle_I$.

Note that the predicates $\langle\!\langle q \rangle\!\rangle_I$, $\langle\!\langle e : q \rangle\!\rangle_I$, $\langle\!\langle q' : q, e \rangle\!\rangle_I$, and $\langle\!\langle \neg e : q \rangle\!\rangle_I$ as well as the encoding for the global disabled transitions can be defined similar to their counterparts discussed earlier.

4 Counterexample Extraction

Our tool specializes the synthesis procedure in Algo. 1 by using QUICKSILVER as the teacher to check phase-compatibility, cutoff-amenability, and safety. For the remainder of this section, we will refer to phase-compatibility and cutoff-amenability conditions as *local* properties and safety (and liveness) specifications as *global* properties.

Algorithm 2: Counterexample Extraction.

procedure Extract(P_I, ϕ)
  $\phi' = makeDNF(\neg\phi)$
  $W = \varnothing$
  foreach $c \in cubes(\phi')$ **do**
    if $[\![P_I]\!] \models c$ **then**
      $cw = \varnothing$
      foreach $l \in literals(c)$ **do**
        $lw = \mathtt{witness}(l)$
        $cw = cw \cup \{lw\}$
      $W = W \cup \{cw\}$
    $cex = pickMinimal(W)$
    return cex

Local Properties. Given a local property ϕ expressed as first-order logic formulas over the local semantics of a MERCURY process, CINNABAR extracts a counterexample cex according to Algo. 2.

First, we negate the property and express in disjunctive normal form (DNF):

$\phi' = \neg\phi = c_1 \vee c_2 \vee \ldots$, where each cube $c_i = l_1 \wedge l_2 \wedge \ldots$ is a conjunction of literals (Line 2). Then, for each cube c satisfied under $[\![P_I]\!]$ (Line 5), extract a cube *witness* cw that is a subset of the local semantics $[\![P_I]\!]$ such that $[\![P_I]\!] \models cw$ (Lines 7 - 9). This is done by extracting, for each literal l in c, a minimal subset lw of $[\![P_I]\!]$ such that $lw \models l$ (Line 8). We say lw is a *minimal witness* of l if any strict subset of lw cannot be a witness for l (i.e., $\forall lw' \subset lw : lw' \not\models l$). Finally pick a minimal (in terms of size) cube witness of some cube c as a cex (Line 11). Since $cex \models c$ and $c \Rightarrow \neg\phi$, we know that $cex \models \neg\phi$ (or equivalently, $cex \not\models \phi$).

In this work, we carefully analyzed the phase-compatibility and cutoff amenability conditions and incorporated procedures to compute witnesses for their literals (i.e., the `witness` calls on Line 8). We refer the interested reader to the extended version [21] of this paper for complete details, and illustrate one such counterexample extraction procedure using an example.

Example. We present a simplified phase-compatibility condition and demonstrate the above procedure on it. Let the set of broadcast, partition, and consensus events be called the *globally-synchronizing* events, denoted $E_{\texttt{global}}$. Let $ph(s)$ be the set of all "phases" containing local state s. The condition states that: for each internal transition $s \to s'$ that is accompanied by a reacting transition $s' \xrightarrow{R(\texttt{f})} s''$ for some globally-synchronizing event $\texttt{f}$, and for each state t in the same phase as s, state t must have a reacting transition of event $\texttt{f}$. Formally:

$$\begin{aligned}&\forall \texttt{f} \in E_{\texttt{global}}, s, s' \in S :\\ &\qquad \big(s \to s' \in T \wedge s' \xrightarrow{R(\texttt{f})} * \in T\big) \Rightarrow \big(\forall X \in ph(s), t \in X :\ \exists t \xrightarrow{R(\texttt{f})} * \in T\big).\end{aligned}$$

This condition is an example of a local property ϕ we want to extract counterexamples for when it fails. The procedure is applied as follows:

Step (1): We first simplify ϕ to the following:

$$\begin{aligned}&\forall \texttt{f} \in E_{\texttt{global}}, s, s', t \in S, X \in ph(s) :\\ &\qquad \big(s \to s' \in T \wedge s' \xrightarrow{R(\texttt{f})} * \in T \wedge inPhase(X, s, t)\big) \Rightarrow \big(\exists t \xrightarrow{R(\texttt{f})} * \in T\big),\end{aligned}$$

where $inPhase(X, s, t)$ indicates that states s and t are in phase X together. We then obtain the negation $\neg\phi$:

$$\begin{aligned}&\exists \texttt{f} \in E_{\texttt{global}}, s, s', t \in S, X \in ph(s) :\\ &\qquad s \to s' \in T \wedge s' \xrightarrow{R(\texttt{f})} * \in T \wedge inPhase(X, s, t) \wedge \neg\exists t \xrightarrow{R(\texttt{f})} * \in T.\end{aligned}$$

Step (2): The formula $\neg\phi$ is in DNF, and there is a cube for each instantiation of event $\texttt{f} \in E_{\texttt{global}}$, states $s, s', t \in S$, and phase X that satisfies the formula $\neg\phi$. There are 4 literals. The literals "$s \to s' \in T$" and "$s' \xrightarrow{R(\texttt{f})} * \in T$" can be witnessed by the corresponding transitions $s \to s'$ and $s' \xrightarrow{R(\texttt{f})} *$, respectively. The literal "$\neg\exists t \xrightarrow{R(\texttt{f})} * \in T$" can be witnessed by the *disabled* transition $t \xrightarrow{R(\texttt{f})} \perp$. The witness for the literal $inPhase(X, s_a, s_b)$ for some phase X and local states s_a and s_b is more involved. It depends on the *nature* of that phase. We analyzed the phase construction procedure given in [19] and distilled it as follows. For each event $e \in E_{\texttt{global}}$, we define its source (resp. destination) set to be the set of states in S from (resp. to) which there exists a transition in T labeled with an acting or reacting action of event e. Let $corePhases$ be the set of all source and destination sets of all globally-synchronizing actions. Then, two states s_a and s_b are in the same phase if:

(a) they are part of some core phase, i.e., $\exists X \in corePhases : s_a, s_b \in X$, or,

(b) they are in different core phases that are connected by an internal path, i.e., $\exists A, B \in corePhases : s_a, s'_a \in A \wedge s_b, s'_b \in B \wedge s'_a \rightsquigarrow s'_b$, where $s'_a \rightsquigarrow s'_b$ is an internal path from s'_a to s'_b.

If X is a core phase (i.e., case (A) holds), the counterexample extraction procedure returns the phase itself. Otherwise, case (B) holds and the two core phases are recursively extracted as well as the internal path connecting them.

Step (3) The final step is to build a subset of the local semantics that include the extracted witnesses for all 4 literals.

Global Properties. If a candidate process P_I meets its phase-compatibility and cutoff-amenability conditions, then it belongs to the efficiently-decidable fragment of MERCURY, and a cutoff c exists. It then remains to check if the system $P_{I,1}||\ldots||P_{I,n}||P_e$ is safe (i.e., $[\![P_I, c]\!] \models \phi_s(c)$).

Safety properties $\phi_s(n)$ are specified by the system designer as (Boolean combinations of) permissible safety specifications. Such properties are invariants that must hold in every reachable state in $[\![P_I, c]\!]$.

A counterexample $cex \subseteq [\![P_I, c]\!]$ to a safety property $\phi_s(c)$ is a finite trace from the initial state q_0 to an error state q_e. Such traces are extracted while constructing $[\![P_I, c]\!]$.

5 Implementation and Evaluation

5.1 Implementation

Our tool, CINNABAR[5], implements the architecture illustrated in Fig. 1. Additionally, it incorporates a liveness checker into the teacher. Liveness properties $\phi_l(c)$ ensure that the system makes progress and eventually reacts to various events. We refer the interested reader to the extended version [21] for details on specifying liveness properties as well as extracting and encoding counterexamples to such properties.

5.2 Evaluation

In this section, we investigate CINNABAR's performance. We study the impact of CINNABAR's counterexample extraction and encoding, as well as the choice of uninterpreted functions, on performance. Finally, we examine how CINNABAR's iterations are distributed across the different types of counterexamples.

[5] CINNABAR is publicly available on Zenodo [20].

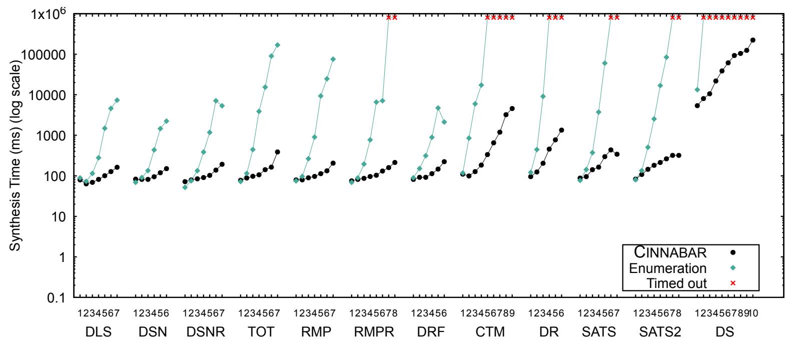

Fig. 2: Cinnabar's performance compared to enumeration-based synthesis. The systems studied are: Distributed Store (DS), Consortium (CTM), Distributed Lock Service (DLS), Distributed Register (DR), Two-Object Tracker (TOT), Distributed Robot Flocking (DRF), variants Small Aircraft Transportation System Landing Protocol (SATS, SATS2), variants of Distributed Sensor Network (DSN, DSNR), and variants of Robotics Motion Planner (RMP, RMPR). For each benchmark, the i-th point denotes the average runtime for all variants with i uninterpreted functions.

Benchmarks. The benchmarks we use are process sketches based on the benchmarks presented in [19]. We refer the reader to the extended version [21] for (i) a description of each benchmark's functionality, its safety and liveness specifications, and the unspecified functionality in the sketch, and (ii) an example Mercury sketch and its completion.

Experimental Setup. To ensure that our reported results are not dependent on a particular choice of uninterpreted functions, we create a set of *variants* for each benchmark as follows. For each benchmark, we first pick a set ue of "candidate uninterpreted functions", corresponding to expressions that a designer might reasonably leave unspecified. Then, for each subset e in the set $\mathcal{P}(ue)$ of all non-empty subsets of ue, we create a variant of the benchmark where the uninterpreted functions in e are included in the sketch. We set a timeout of 15 minutes when running any variant and conduct our experiments on a MacBook Pro with 2 GHz Quad-Core Intel Core i5 and 16 GB of RAM.

Effect of Counterexample Extraction and Encoding. As our baseline, we consider a synthesis loop where the learner enumerates interpretations until a correct interpretation is found. If some interpreted process sketch P_I fails a property at any stage, we add the constraint $\texttt{c} = \neg I$ to the learner. This effectively eliminates one interpretation at a time, as opposed to all interpretations that exhibit the given counterexample at a time (as done by our encoder). In Fig. 2, we present a comparison of Cinnabar's runtime compared to this enumeration-based baseline. We make the following observations. While the runtimes of both enumeration-based synthesis and Cinnabar grow exponentially when increasing the number of uninterpreted functions, Cinnabar outperforms

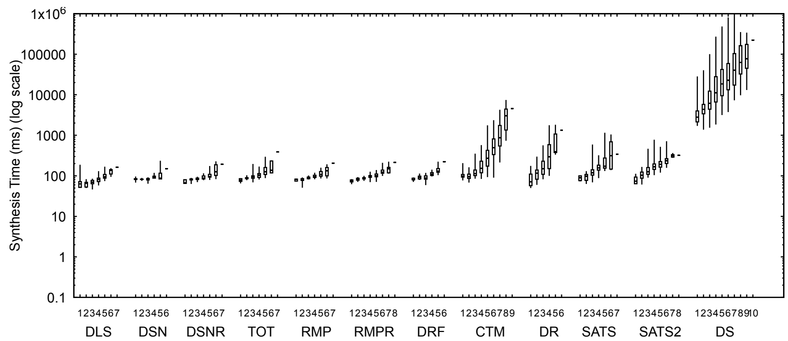

Fig. 3: Effect of the choice of uninterpreted functions on synthesis time. For some benchmark and some number m of uninterpreted functions, the m-th box-and-whiskers plot presents, from bottom to top, the minimum, first quartile, median, third quartile, and maximum synthesis run time across the run times of all variants of that benchmark with m uninterpreted functions.

enumeration-based synthesis in almost all scenarios. Only for variants with a single uninterpreted function we observed cases where enumeration-based synthesis found a correct solution faster than Cinnabar (e.g., as in DSNR with one uninterpreted function). This is due to the additional time spent extracting and encoding counterexamples. However, the value of the counterexample extraction and encoding becomes clearly apparent with larger number of unspecified expressions as the number of interpretations grows much larger and it becomes infeasible to just enumerate them. Furthermore, Cinnabar is able to perform synthesis for any variant of our benchmarks in under 9 minutes.

Effect of the Choice of Uninterpreted Functions. In Fig. 3, for each benchmark, we examine the variation of synthesis runtime across variants with the same number of uninterpreted functions. As shown in the figure, in some cases (e.g., CTM and DS), the variation is more noticeable. The main factor contributing to this is that uninterpreted functions present different overhead on synthesis based on their nature. For instance, an uninterpreted function corresponding to a `lhs` of some assignment expression is more expensive to synthesize compared to an uninterpreted function corresponding to a target of some `goto` statement, as the latter has a smaller search space.

Counterexample Distribution on Iterations. In Fig. 4, we illustrate the different types of counterexamples encountered throughout Cinnabar's iterations. We make the following observations. First, Cinnabar spends most of its iterations ruling out phase-compatibility violations. This is expected as checking phase-compatibility is the first stage in our synthesis loop. Since a phase-compatible system moves in a structured way between its phases, this stage rules out all arbitrary completions that prohibit processes from advancing through the phases. Furthermore, there are fewer safety violations than any other type of violations. Once an interpreted process sketch is in the efficiently-decidable fragment

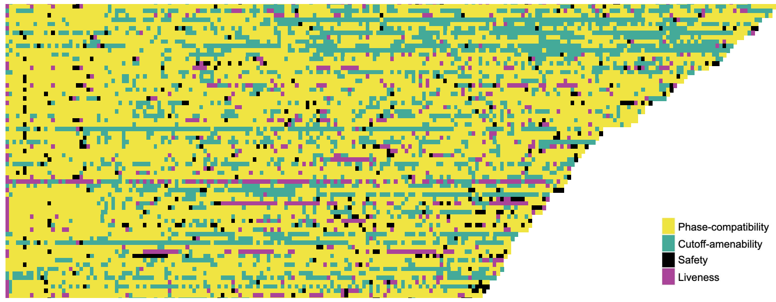

Fig. 4: A property-based visualization of CINNABAR's iterations for a representative subset of the variants. Each line corresponds a CINNABAR's execution of a synthesis variant of a benchmark. From left to right, each line starts with iteration 1, ends with the iteration where a correct interpretation was found, and is colored to indicate nature of violations encountered throughout the execution. For instance, the line would indicate that CINNABAR encountered a phase-compatibility violation in iteration 1, then a cutoff-amenability in iteration 2, ..., and finally was able to find a correct interpretation in iteration 6.

of MERCURY, it is more likely to be safe. There are two factors that contribute to this: (i) phase-compatible systems move in a structured way and are more likely to be "closer" to a correct version of the system, and (ii) because cutoff-amenability depends on the safety specification, satisfying cutoff-amenability means the interpreted process sketch is more likely to be correct with respect to the safety property already. Finally, eliminating liveness violations ensures that CINNABAR is able to synthesize higher-quality completions. As shown in the figure, liveness violations are often encountered in the very first iteration, as the SMT-based learner tends to favor interpretations with disabled guards that trivially satisfy phase-compatibility, cutoff-amenability, and safety properties.

Usability. If CINNABAR fails to synthesize a correct completion, the designer can *replace* existing expressions in the sketch with uninterpreted functions, allowing CINNABAR to explore a larger set of possible candidate completions.

Finally, while the supported uninterpreted functions may not correspond to large segments of the code or complex control-flow constructs, they are the main "knobs" that the designer needs to turn to ensure that their systems belong to the efficiently-decidable fragment of MERCURY.

6 Related Work

Aiding System Designers via Decidable Verification. Ivy [31] adopts an interactive approach to aid the designer in searching for inductive invariants for their systems. Ivy translates the system model and its invariant to EPR [32], and looks for a *counterexample-to-induction* (CTI). The designer adjusts the invariant to eliminate that CTI and Ivy starts over. I4 [28] builds on Ivy by first considering a fixed system size, automatically generating a potential inductive invariant, and using Ivy to check if that invariant is also valid for any system size. The approach in [12] identifies a class of asynchronous systems that can be reduced to an equivalent synchronized system modeled in the Heard-Of Model [10]. The designer manually annotates the asynchronous system to facilitate the reduction, and encodes the resulting Heard-Of model in the $\mathbb{CL}$ [15] logic which has a semi-decision procedure. These approaches differ from ours in two ways. First, the designer needs to manually provide/manipulate inductive invariants and/or annotations to eventually enable decidable verification. Second, these approaches are "verification only": they require a *fully-specified* model that either meets or violates its correctness properties and the designer is responsible for adjusting the model if verification fails. CINNABAR, on the other hand, accepts a sketch that is then completed to meet its properties.

Parameterized Synthesis. Jacobs and Bloem [22] introduced a general approach for parameterized synthesis based on cutoffs, where they use an underlying fixed-size synthesis procedure that is required to guarantee that the conditions for cutoffs are met by the synthesized implementation. Our approach can be seen as an instantiation of this approach, as one of the stages in our multi-stage counterexample-based loop ensures that cutoff-amenability conditions hold on any candidate process. Other approaches that tackle the parameterized synthesis problem without cutoff results are more specialized. For instance, the approach in [26] adopts a CEGIS-based synthesis strategy where the designer provides a threshold automaton with some parameters unspecified. Synthesis completes the model and uses the parameterized model checker in [25] to check the system. A similar idea, but based on the notion of well-structured transition systems, is used for the automatic *repair* of parameterized systems in [23]. The approach in [24] targets parameterized synthesis for self-stabilizing rings, and shows that the problem is decidable even when the corresponding parameterized verification problem is not. The designer provides a set of legitimate states and the size of the template process, and the procedure yields a completed self-stabilizing template. A similar approach for more general topologies is presented in [30]. Bertrand et al. [6] target systems composed of an unbounded number of agents that are fully specified and one underspecified controller process. The synthesis goal is to synthesize a controller that controls all agents uniformly and guides them to a specific desired state. Markgraf et al. [29] also target synthesis of controllers by posing the problem as an infinite-duration 2-player game and utilize regular model checking and the L* algorithm [4] to learn correct-by-design controllers. These approaches are not applicable to our setup as they do not admit distributed agreement-based systems (modeled in MERCURY).

Synthesis of Distributed Systems with a Fixed Number of Processes. Various approaches focus on automated synthesis of distributed systems with a *fixed* number of processes [3,2,1,13,37]. While such approaches deploy a similar counterexample-guided strategy to complete a user-provided sketch, they do not provide parameterized correctness guarantees nor the necessary agreement primitives needed to model distributed agreement-based systems.

References

1. Alur, R., Martin, M., Raghothaman, M., Stergiou, C., Tripakis, S., Udupa, A.: Synthesizing finite-state protocols from scenarios and requirements. In: Yahav, E. (ed.) Hardware and Software: Verification and Testing. pp. 75–91. Springer International Publishing, Cham (2014)
2. Alur, R., Raghothaman, M., Stergiou, C., Tripakis, S., Udupa, A.: Automatic completion of distributed protocols with symmetry. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. pp. 395–412. Springer International Publishing, Cham (2015)
3. Alur, R., Tripakis, S.: Automatic synthesis of distributed protocols. SIGACT News **48**(1), 55–90 (Mar 2017). https://doi.org/10.1145/3061640.3061652, https://doi.org/10.1145/3061640.3061652
4. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (nov 1987). https://doi.org/10.1016/0890-5401(87)90052-6, https://doi.org/10.1016/0890-5401(87)90052-6
5. Apt, K.R., Kozen, D.C.: Limits for automatic verification of finite-state concurrent systems. Information Processing Letters **22**(6), 307–309 (1986). https://doi.org/https://doi.org/10.1016/0020-0190(86)90071-2, https://www.sciencedirect.com/science/article/pii/0020019086900712
6. Bertrand, N., Dewaskar, M., Genest, B., Gimbert, H., Godbole, A.A.: Controlling a population. arXiv preprint arXiv:1807.00893 (2018)
7. Bloem, R., Jacobs, S., Khalimov, A., Konnov, I., Rubin, S., Veith, H., Widder, J.: Decidability of Parameterized Verification. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers (2015)
8. Bornholt, J., Joshi, R., Astrauskas, V., Cully, B., Kragl, B., Markle, S., Sauri, K., Schleit, D., Slatton, G., Tasiran, S., Van Geffen, J., Warfield, A.: Using lightweight formal methods to validate a key-value storage node in amazon s3. In: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. p. 836–850. SOSP '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3477132.3483540, https://doi.org/10.1145/3477132.3483540
9. Büchi, J.R.: On a Decision Method in Restricted Second Order Arithmetic, pp. 425–435. Springer New York, New York, NY (1990). https://doi.org/10.1007/978-1-4613-8928-6_23, https://doi.org/10.1007/978-1-4613-8928-6_23
10. Charron-Bost, B., Schiper, A.: The Heard-of Model: Computing in Distributed Systems with Benign Faults. Distributed Computing **22**(1), 49–71 (2009). https://doi.org/10.1007/s00446-009-0084-6
11. Cook, B.: Formal reasoning about the security of amazon web services. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification. pp. 38–47. Springer International Publishing, Cham (2018)
12. Damian, A., Dragoi, C., Militaru, A., Widder, J.: Communication-closed Asynchronous Protocols. In: International Conference on Computer Aided Verification (2019)
13. Damm, W., Finkbeiner, B.: Automatic Compositional Synthesis of Distributed Systems. In: International Symposium on Formal Methods. pp. 179–193. Springer (2014)
14. Dill, D., Grieskamp, W., Park, J., Qadeer, S., Xu, M., Zhong, E.: Fast and reliable formal verification of smart contracts with the move prover. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 183–200. Springer International Publishing, Cham (2022)

15. Drăgoi, C., Henzinger, T.A., Veith, H., Widder, J., Zufferey, D.: A Logic-based Framework for Verifying Consensus Algorithms. In: International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 161–181. Springer (2014)
16. Emerson, E.A., Sistla, A.P.: Utilizing symmetry when model-checking under fairness assumptions: An automata-theoretic approach. ACM Trans. Program. Lang. Syst. **19**(4), 617–638 (jul 1997). `https://doi.org/10.1145/262004.262008`, `https://doi.org/10.1145/262004.262008`
17. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S., Zill, B.: Ironfleet: Proving practical distributed systems correct. In: Proceedings of the 25th Symposium on Operating Systems Principles. p. 1–17. SOSP '15, Association for Computing Machinery, New York, NY, USA (2015). `https://doi.org/10.1145/2815400.2815428`, `https://doi.org/10.1145/2815400.2815428`
18. Jaber, N., Jacobs, S., Wagner, C., Kulkarni, M., Samanta, R.: Parameterized verification of systems with global synchronization and guards. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification. pp. 299–323. Springer International Publishing, Cham (2020)
19. Jaber, N., Wagner, C., Jacobs, S., Kulkarni, M., Samanta, R.: Quicksilver: Modeling and parameterized verification for distributed agreement-based systems. Proc. ACM Program. Lang. **5**(OOPSLA) (oct 2021). `https://doi.org/10.1145/3485534`, `https://doi.org/10.1145/3485534`
20. Jaber, N., Wagner, C., Jacobs, S., Kulkarni, M., Samanta, R.: Synthesis of Distributed Agreement-Based Systems with Efficiently-Decidable Verification (Artifact) (Apr 2023). `https://doi.org/10.5281/zenodo.7497463`, `https://doi.org/10.5281/zenodo.7497463`
21. Jaber, N., Wagner, C., Jacobs, S., Kulkarni, M., Samanta, R.: Synthesis of distributed agreement-based systems with efficiently-decidable verification (extended version) (2023). `https://doi.org/10.48550/ARXIV.2208.12400`, `https://arxiv.org/abs/2208.12400`
22. Jacobs, S., Bloem, R.: Parameterized Synthesis. Logical Methods in Computer Science **10**(1) (2014)
23. Jacobs, S., Sakr, M., Völp, M.: Automatic repair and deadlock detection for parameterized systems. In: FMCAD 2022. pp. 225–234
24. Klinkhamer, A.P., Ebnenasir, A.: Synthesizing parameterized self-stabilizing rings with constant-space processes. In: Dastani, M., Sirjani, M. (eds.) Fundamentals of Software Engineering. pp. 100–115. Springer International Publishing, Cham (2017)
25. Konnov, I., Lazić, M., Veith, H., Widder, J.: A Short Counterexample Property for Safety and Liveness Verification of Fault-tolerant Distributed Algorithms. ACM SIGPLAN Notices **52**(1), 719–734 (2017)
26. Lazic, M., Konnov, I., Widder, J., Bloem, R.: Synthesis of Distributed Algorithms with Parameterized Threshold Guards. In: Aspnes, J., Bessani, A., Felber, P., Leitão, J. (eds.) OPODIS. LIPIcs, vol. 95, pp. 32:1–32:20. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
27. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning. pp. 348–370. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
28. Ma, H., Goel, A., Jeannin, J.B., Kapritsos, M., Kasikci, B., Sakallah, K.A.: I4: Incremental inference of inductive invariants for verification of distributed proto-

cols. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles. p. 370–384. SOSP '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3341301.3359651, https://doi.org/10.1145/3341301.3359651
29. Markgraf, O., Hong, C.D., Lin, A.W., Najib, M., Neider, D.: Parameterized synthesis with safety properties. In: Oliveira, B.C.d.S. (ed.) Programming Languages and Systems. pp. 273–292. Springer International Publishing, Cham (2020)
30. Mirzaie, N., Faghih, F., Jacobs, S., Bonakdarpour, B.: Parameterized synthesis of self-stabilizing protocols in symmetric networks. Acta Informatica **57**(1-2), 271–304 (2020)
31. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: Safety verification by interactive generalization. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 614–630. PLDI '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2908080.2908118, https://doi.org/10.1145/2908080.2908118
32. Piskac, R., de Moura, L., Bjørner, N.: Deciding Effectively Propositional Logic Using DPLL and Substitution Sets. Journal of Automated Reasoning **44**(4), 401–424 (2010)
33. Reid, A., Flur, S., Church, L., de Haas, S., Johnson, M., Laurie, B.: Towards making formal methods normal: meeting developers where they are. In: HATRA 2020: Human Aspects of Types and Reasoning Assistants (2020), https://arxiv.org/abs/2010.16345
34. Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and proving with distributed protocols. Proc. ACM Program. Lang. **2**(POPL) (Dec 2017). https://doi.org/10.1145/3158116, https://doi.org/10.1145/3158116
35. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial Sketching for Finite Programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 404–415. ASPLOS XII, ACM (2006)
36. Suzuki, I.: Proving properties of a ring of finite-state machines. Inf. Process. Lett. **28**(4), 213–214 (Jul 1988). https://doi.org/10.1016/0020-0190(88)90211-6, https://doi.org/10.1016/0020-0190(88)90211-6
37. Udupa, A., Raghavan, A., Deshmukh, J.V., Mador-Haim, S., Martin, M.M., Alur, R.: TRANSIT: Specifying Protocols with Concolic Snippets. ACM SIGPLAN Notices **48**(6), 287–296 (2013)
38. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.: Verdi: A framework for implementing and formally verifying distributed systems. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 357–368. PLDI '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2737924.2737958, https://doi.org/10.1145/2737924.2737958