QuickSilver: Modeling and Parameterized Verification for Distributed Agreement-Based Systems

NOURALDIN JABER, Purdue University, USA
CHRISTOPHER WAGNER, Purdue University, USA
SWEN JACOBS, CISPA Helmholtz Center for Information Security, Germany
MILIND KULKARNI, Purdue University, USA
ROOPSHA SAMANTA, Purdue University, USA

The last decade has sparked several valiant efforts in deductive verification of distributed agreement protocols such as consensus and leader election. Oddly, there have been far fewer verification efforts that go beyond the core protocols and target applications that are *built on top of* agreement protocols. This is unfortunate, as agreement-based distributed services such as data stores, locks, and ledgers are ubiquitous and potentially permit modular, scalable verification approaches that mimic their modular design.

We address this need for verification of distributed agreement-based systems through our novel modeling and verification framework, QuickSilver, that is not only modular, but also fully automated. The key enabling feature of QuickSilver is our encoding of *abstractions of verified agreement protocols* that facilitates modular, decidable, and scalable automated verification. We demonstrate the potential of QuickSilver by modeling and efficiently verifying a series of tricky case studies, adapted from real-world applications, such as a data store, a lock service, a surveillance system, a pathfinding algorithm for mobile robots, and more.

CCS Concepts: • **Theory of computation → Program verification**; **Distributed computing models**; **Automated reasoning**; **Verification by model checking**; *Abstraction*; *Concurrency*; *Program analysis*.

Additional Key Words and Phrases: Parameterized Verification, Modular Verification, Distributed Systems

ACM Reference Format:
Nouraldin Jaber, Christopher Wagner, Swen Jacobs, Milind Kulkarni, and Roopsha Samanta. 2021. QuickSilver: Modeling and Parameterized Verification for Distributed Agreement-Based Systems. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 157 (October 2021), 31 pages. https://doi.org/10.1145/3485534

1 INTRODUCTION

Modern distributed services such as data stores, logs, caches, queues, locks, and ledgers heavily rely on *distributed agreement* to perform their higher-level functions—processes in these distributed services need to agree on a leader, on the members of a group, on configurations, or on owners of locks. Notable instances of such *distributed agreement-based* services include the Chubby lock service [Burrows 2006] and RedisRaft key-value store [RedisRaft 2021], which are built on top of the Paxos [Lamport 1998] and Raft [Ongaro and Ousterhout 2014] consensus algorithms, respectively. The importance of agreement protocols as a key building block in distributed services has sparked significant verification efforts for these protocols [Chand et al. 2016; Cousineau et al. 2012; Drăgoi

Authors' addresses: Nouraldin Jaber, Purdue University, West Lafayette, USA, njaber@purdue.edu; Christopher Wagner, Purdue University, West Lafayette, USA, wagne279@purdue.edu; Swen Jacobs, CISPA Helmholtz Center for Information Security, Saarbrücken, Germany, jacobs@cispa.saarland; Milind Kulkarni, Purdue University, West Lafayette, USA, milind@purdue.edu; Roopsha Samanta, Purdue University, West Lafayette, USA, roopsha@purdue.edu.

2475-1421/2021/10-ART157
https://doi.org/10.1145/3485534

et al. 2014; Drăgoi et al. 2016; García-Pérez et al. 2018; Lamport 2002; Liu et al. 2012; Marić et al. 2017; Padon et al. 2017b; Woos et al. 2016]. Intriguingly, with rare exceptions, these efforts restrict their attention to the core protocols and do not consider the distributed services that *build* on those protocols. This is unfortunate because there are arguably more distributed systems that build on agreement protocols than there are implementations of these core protocols. Moreover, such agreement-based systems are more likely to be developed by non-experts who can benefit from verification. In this paper, we ask *can we develop modular modeling and verification frameworks for distributed agreement-based systems* by (1) assuming that the underlying agreement protocols are verified separately and (2) encapsulating their complexities within cleanly-defined abstractions? Such an approach would both allow us to leverage the heroic efforts towards verifying agreement protocols as well as ease the burden of modeling the distributed systems that rely on those protocols. We note that existing verification efforts for agreement-based systems that go beyond core protocols [Hawblitzel et al. 2015; Liu et al. 2012; Padon et al. 2016; v. Gleissenthall et al. 2019], with the exception of [Griffin et al. 2020; Sergey et al. 2017], do not leverage the availability of verified agreement artifacts through systematic agreement abstractions.

We further ask: can our agreement abstractions enable *fully automated, parameterized verification* for interesting classes of agreement-based systems? This second question is an open one. The parameterized model checking problem (PMCP)—the problem of algorithmically verifying correctness of systems parameterized by the number of processes—is well-known to be undecidable in its full generality [Apt and Kozen 1986; Suzuki 1988]. While decidability has been shown for some restricted classes of distributed systems, it is unclear whether agreement-based systems allow for a decidable parameterized verification procedure at all. Past verification efforts for agreement protocols/implementations as well as agreement-based systems sidestep the decidability issue by preferring the use of interactive or semi-automated deductive verification over model checking. The appeal of *push-button* verification that does not require a user to provide inductive invariants or manipulate a theorem prover, however, remains undeniable. We argue that abstracting away and separately verifying the intricate details of agreement (using deductive techniques) should yield *simpler* models of agreement-based systems that may *now* become amenable to decidable and scalable model checking.

In this paper, we propose the QuickSilver framework for modeling and parameterized model checking of distributed agreement-based systems. QuickSilver advances a brand new verification strategy for agreement-based systems that is not only modular, but also fully automated.

1.1 The QuickSilver Framework

In our design of QuickSilver, we address several questions:

(1) *How should we abstract agreement?* The primitives we develop to abstract agreement must be sufficiently general to capture the essential characteristics of a wide variety of agreement protocols, while still permitting decidable parameterized model checking of distributed systems with such primitives.
(2) *How should we model our systems?* The modeling language we use for distributed agreement-based systems should match the manner in which system designers build their programs.
(3) *How should we identify systems that enable decidable and scalable verification?* The fundamental obstacle we need to tackle is the undecidability of PMCP. Thus, we must find *easily-checkable* conditions under which the verification of systems we model (including their use of agreement primitives) is decidable. Further, because our goal is to model fairly complex systems, we must endeavor to find scalable approaches for verifying these systems.

In particular, QuickSilver makes the following contributions.

Mercury: A Modeling Language with Agreement Primitives. We carefully examined a range of agreement protocols in the literature, such as consensus and leader election, and observed that while the protocol internals differed substantially, their externally-observable behavior could be captured with two *agreement primitives* (namely, **`Partition`** and **`Consensus`**) that have simple semantics and abstract away the protocols' implementation details. The **`Partition`** primitive allows a set of participant processes to divide themselves into groups (e.g., leaders and followers). The **`Consensus`** primitive allows its participants, with each proposing a value, to agree on a finite set of decided values. Sec. 3 presents a new, intuitive modeling language, Mercury[1], that allows designers to model finite-state distributed systems using these agreement primitives, and hence design systems without worrying about the internals of the core agreement protocols.

Parameterized Verification of Mercury Systems. With Mercury's primitives abstracting away the messy details of distributed agreement, we observe that the resulting higher-level systems can be more amenable to automated verification. Sec. 4 identifies a broad class of Mercury systems that permit decidable and efficient parameterized verification. In particular, we present two key results. First, we identify syntactic conditions on Mercury systems that yield decidability of PMCP. Second, we identify additional syntactic conditions that, for a given class of safety/reachability properties, enable *practical* parameterized verification by providing *cutoffs*: a number k of processes such that verifying the correctness of a *fixed-size* k-process system implies the correctness of arbitrary-sized systems. This result means that *non-parameterized* model checkers can be leveraged to provide parameterized verification.

We prove both results by (1) defining Mercury Core, a novel extension of the decidable and cutoff-yielding fragments of a recently proposed abstract model for distributed systems [Jaber et al. 2020a] and (2) showing that Mercury systems satisfying our syntactic conditions are *simulation equivalent* to systems in Mercury Core.

The class of safety/reachability properties to which these results apply include properties forbidding the reachability of global states where *more than a fixed number* of processes are simultaneously in some pivotal local states. An example of such a property is mutual exclusion of a certain critical local state, i.e., *no more than two* processes can reach the critical state simultaneously in any execution. These results currently do not apply to liveness properties, e.g., a leader is eventually elected, or to arbitrary safety properties, e.g., there exists *at least one leader* at all times, or, *no more than half of the processes* can simultaneously be leaders.

Implementation, Mercury Benchmarks, and Evaluation. With our decidability and cutoff results for Mercury programs in hand, we have an approach that enables scalable, parameterized verification of distributed agreement-based systems *in theory*. Sec. 5 instantiates the theoretical results of QuickSilver by presenting an implementation of a cutoff-driven parameterized verification procedure for Mercury systems. Crucially, QuickSilver *automatically* checks the syntactic conditions that yield practical parameterized verification. When a system is not practically verifiable, QuickSilver provides best-effort feedback suggesting modifications to the system that may make it so. We show that complex distributed agreement-based systems including a data store, a lock service, a surveillance system, a pathfinding algorithm for mobile robots, the Small Aircraft Transportation System (SATS) protocol [NASA 2021], and several other interesting applications can be naturally and succinctly modeled in Mercury, and can then be efficiently verified.

1.2 Related Work

We first compare with the most related lines of work.

[1]Modeling Event Reaction and Coordination Using symmetRY

Global Synchronization Protocols (GSPs). In recent work, Jaber et al. [2020a] propose a new model, GSP, for crash- and failure-free distributed systems and present decidability and cutoff results for parameterized verification of systems in the model. This model supports global transitions associated with global guards which can be used by multiple processes to synchronize collectively and simultaneously. Such global transitions and guards can be used, *in theory*, to carefully encode abstractions of agreement protocols. However, the GSP model is an abstract, theoretical model based on counter abstraction and does not provide an intuitive, accessible interface for system designers; for instance, processes in the GSP model cannot use local variables and are specified as low-level state-transition systems with *manually-inferred* guards. Additionally, users are required to *manually check* if their GSP system models fall within the decidable, cutoff-yielding fragment.

In contrast, QuickSilver (i) provides a user-friendly modeling language for distributed systems with inbuilt primitives that are designed to abstract agreement protocols, (ii) supports process crash-stop failures, (iii) pushes the boundaries of decidable parameterized verification by expanding the GSP decidability fragment, and (iv) includes a *fully-automated implementation* for checking if Mercury programs belong to the expanded decidable, cutoff-yielding fragment.

Modular Verification with Abstract Modules. Disel [Sergey et al. 2017] and TLC [Griffin et al. 2020] leverage the same observation we do—that distributed applications build on standard protocols—and enable users to incorporate abstractions of such protocols to provide modular verification using the Coq theorem prover. The user is responsible for providing both the high-level descriptions of the underlying protocols as well as the inductive invariants needed to link protocols to their clients and/or enable *horizontal composition* with other protocols. The TLC framework could potentially reason about agreement-based systems as it supports *vertical composition*, but the user would need to manually incorporate abstractions of the underlying agreement protocols. In contrast, QuickSilver is equipped with intuitive, inbuilt primitives that abstract agreement protocols and facilitate vertical composition and fully-automated parameterized verification.

In what follows, we discuss other broad themes of verification approaches for distributed systems.

Semi-Automated, Deductive Verification. Approaches for semi-automated, deductive verification of distributed protocols and implementations expect a user to specify inductive invariants [Andersen and Sergey 2019; Doenges et al. 2017; Feldman et al. 2019; Krogh-Jespersen et al. 2020; Padon et al. 2016; Rahli 2012; Sergey et al. 2017; Wilcox et al. 2017, 2015; Woos et al. 2016]. Some approaches [Damian et al. 2019; Padon et al. 2017a,b; Taube et al. 2018] enable more (but not full) automation by translating the user-provided system and inductive invariants into a decidable fragment of first-order logic (e.g., effectively propositional logic (EPR) [Piskac et al. 2010]) or a model with a semi-automatic verification procedure (e.g., the Heard-Of model [Charron-Bost and Schiper 2009]). Recent work [Kragl et al. 2020; v. Gleissenthall et al. 2019] proposes the use of Lipton's reduction [Lipton 1975] to reduce reasoning about asynchronous programs to synchronous and sequential programs, respectively, thereby greatly simplifying the invariants needed. Our approach builds on deductive verification for agreement protocols to enable modeling and automated parameterized verification of systems built on top of verified agreement protocols.

Model Checking. Prior work on PMCP identifies decidable fragments based on restrictions on the communication primitives, specifications, and structure of the system [Aminof et al. 2018; Außerlechner et al. 2016; Delzanno et al. 2002; Emerson and Kahlon 2003b; Esparza et al. 1999; German and Sistla 1992; Jaber et al. 2020a]. To enable efficient parameterized verification, prior work additionally identifies cutoff results for various classes of systems, e.g., cache coherence protocols [Emerson and Kahlon 2003a], guarded protocols [Jacobs and Sakr 2018], consensus protocols [Marić et al. 2017], and self-stabilizing systems [Bloem et al. 2016]. Unfortunately, no existing decidability and cutoff results, except for those in [Jaber et al. 2020a], extend to agreement-based systems.

There has also been some work on model checking and synthesis of distributed systems with a *fixed* number of finite-state processes [Alur et al. 2014, 2015; Alur and Tripakis 2017; Damm and Finkbeiner 2014; Liu et al. 2012; Yang et al. 2009]. However, these frameworks are not naturally extendable for parameterized reasoning and do not consider abstractions of agreement protocols for improving scalability of verification in the fixed-size setting.

2 QUICKSILVER OVERVIEW

This section presents an illustrative example of a complex system that leverages multiple instances of distributed agreement for its high-level function. It then provides an overview of the key building blocks of our modeling language and verification approach using the example. We begin the section with a brief review of distributed agreement protocols.

Distributed Agreement Protocols. Distributed agreement protocols enable a set of distributed participants, each proposing one value, to *collectively decide* on a set of proposals in the presence of failures and asynchrony. There are many variants of agreement protocols with small differences in their decision objectives. For instance, the participants may wish to decide on a single proposal [Lamport 1998, 2006; Mao et al. 2008], an infinite sequence of proposals [Chandra et al. 2007; Ongaro and Ousterhout 2014], or a finite set of leaders amongst themselves [Arghavani et al. 2011; Garcia-Molina 1982]. Despite these variations, any *correct* agreement protocol is characterized by the following three guarantees [Lynch 1996]: (i) *agreement*—all participants decide on the same set of proposals, (ii) *validity*—every proposal in the decided set of proposals must have been proposed by a participant, and (iii) *termination*—all participants eventually decide. Accordingly, recent work in verification of agreement protocols and/or their implementations focuses on guaranteeing agreement, validity, and termination (or, some reasonable variant of these properties).

2.1 Illustrative Example: Distributed Store

Suppose a system designer wants to model and verify a distributed store where multiple processes consistently replicate and update a piece of stored data in response to client requests. The clients may request various operations on the stored data including read and update. To ensure that the data is consistent across all replicas, the designer decides to use distributed agreement protocols to determine which operation these replicas should execute next. For efficiency, they use a leader election protocol to pick a leader that acts as a single point of contact to handle requests from the clients. These requests are replicated to all other processes using a consensus algorithm to maintain consistent stored data throughout the system. This design pattern is common in distributed services like fault-tolerant key-value stores (e.g., RedisRaft [2021]). The safety properties for this system are: (1) there is at most one leader at any given time and (2) all processes agree on the stored data.

Notice that the designer's scheme for the distributed store uses different agreement protocols as building blocks and is inherently *modular*. Moreover, the safety properties are about the high-level design of the distributed store and do not refer to the internals of the leader election and consensus protocols used. Hence, it is sensible to also adopt a modular approach to reasoning about the correctness of the design. Specifically, instead of reasoning about the distributed store as a monolithic program with all agreement protocols modeled explicitly, one can assume that the underlying agreement protocols are verified separately and verify the system where these protocols are replaced with simpler abstractions that capture their behaviors.

Our framework, QuickSilver, and its modeling language Mercury (presented in Sec. 3) enables the designer to utilize such a modular verification approach with the agreement protocols represented using special primitives (denoted **`Partition`** and **`Consensus`** for leader election and consensus, respectively) that soundly abstract the semantics of agreement protocols.

```
1  process DistributedStore
2  variables
3      int[1,5] cmd := 1
4      int[1,2] stored := 1
5  actions
6    env
7      rz doCmd : int[1,5]
8      rz ackCmd : int[1,5]
9      rz ret : int[1,2]
10     br LeaderDown : unit
11
12 initial location Candidate
13   on Partition<elect>(All,1)
14     win: goto Leader
15     lose: goto Replica
16
17 location Leader
18   on recv(doCmd) do
19     cmd := doCmd.payld
20     if(cmd <= 2 && stored != cmd)
21       goto RepCmd
22     else if(cmd = 3)
23       sendrz(ret[stored], doCmd.sID)
24     else
25       goto RepCmd
26 location RepCmd
27   on Consensus<vcCmd>(All,1,cmd) do
28     cmd := vcCmd.decVar[1]
29     if(cmd <= 2) /*set*/
30       stored := cmd
31     else if(cmd = 4) /*inc*/
32       stored := stored + 1
33     else /*dec*/
34       stored := stored - 1
35     sendrz(ackCmd[cmd], doCmd.sID)
36     goto Leader
37
38 location Replica
39   on Consensus<vcCmd>(All,1,_) do
40     cmd := vcCmd.decVar[1]
41     if(cmd <= 2) /*set*/
42       stored := cmd
43     else if(cmd = 4) /*inc*/
44       stored := stored + 1
45     else /*dec*/
46       stored := stored - 1
47   on recv(LeaderDown) do
48     goto Candidate
```

Safety Property: In every reachable state, there is at most one leader.
Safety Property: In every reachable state, all processes in locations `Replica` and `Leader` agree on the value of the variable `stored`.

Fig. 1. Mercury Representation of a Distributed Store Process Definition. A process in a Mercury program consists of a collection of variables, communication actions, and locations with associated event handlers. Each event handler consists of an event and a reaction to that event. An event is an empty event, a receive of a communication action, or one of the two agreement primitives: **`Partition`** and **`Consensus`**. Reactions typically consist of a block of update statements, control statements, and/or sends of communication actions.

For instance, the designer may model their distributed store in Mercury as shown in (Fig. 1). Processes start in the `Candidate` location (Line 12) and coordinate with each other (Line 13) to elect one leader to move to the `Leader` location (Line 17), while the remaining processes become replicas and go to the `Replica` location (Line 38). The leader can receive requests from clients (via the `doCmd` message on Line 18), while the replicas wait for the leader to replicate requests to them.

When the leader receives a request from a client (which could be one of several potential commands), it handles the request on lines 19–25. A command payload consists of either a directive to set the value to 1 or 2 (`cmd <= 2`); to read the stored value (`cmd = 3`); or to increment (`cmd = 4`) or decrement (`cmd = 5`) the stored value. On a read request, the leader responds by returning its stored data to that client (Line 22). When the leader receives any update request, such as a request to set, increment, or decrement its stored data, the leader moves to the `RepCmd` location to initiate a round of consensus to replicate the operation to all replicas in the system (Line 27 for the leader, Line 39 for the replicas). When consensus is complete, all processes update their stored data by executing the operation on which they have agreed, and the leader returns an acknowledgment message to the requesting client. In the event that the leader process has crashed (modeled by

the environment sending a special `LeaderDown` message, not shown in Fig. 1), all processes in the `Replica` location receive that message (Line 47) and return to the `Candidate` location so a new leader may be elected.

Note that the key feature of Mercury's design arises from the encapsulation of the two distributed agreement operations that occur: choosing a leader (Line 13), captured by the **`Partition`** primitive, and that leader's replicating commands to the replicas (lines 27 and 39), captured by the **`Consensus`** primitive. These primitives have carefully-designed semantics that capture the essence of agreement protocols, and allow Mercury programs to be built without considering how that agreement is implemented. Sec. 2.2 describes these primitives and motivates their design in more detail.

Having represented the distributed store in Mercury, the designer can now utilize QuickSilver's push-button parameterized verifier to check the correctness of any distributed system consisting of one or more such identical processes with respect to the two safety properties. In particular, QuickSilver can automatically verify that this Mercury distributed store satisfies the safety properties regardless of the number of processes, in less than a minute. Moreover, any refinement of this program that instantiates the **`Partition`** and **`Consensus`** primitives with some verified leader election or consensus protocol, respectively, is also guaranteed to satisfy the safety properties.

2.2 Agreement Primitives

The design of Mercury's agreement primitives is driven by our goal of automated, parameterized verification of agreement-based systems. Thus, the granularity of abstraction in the agreement primitives was carefully chosen to strike a balance between (a) capturing the *essence* of most practical agreement protocols without modeling protocol-specific behavior and (b) facilitating decidability of PMCP (when combined with additional syntactic conditions). To meet these objectives, we propose *two* agreement primitives, **`Partition`** and **`Consensus`**, that can individually model two common variants of agreement that we refer to as *partition-and-move* agreement and *value-consensus* agreement, respectively. The two primitives can further be composed together to model other variants of agreement. We informally explain these primitives here, and provide a more formal treatment of their semantics in Sec. 3.3.

The* `Partition` *Primitive. The **`Partition`** agreement primitive is used to model partition-and-move agreement, where a set of participants wishes to partition itself into groups. Instances of partition-and-move agreement include variants of leader election protocols which partition the participants into two groups: leaders (or, *winners*) and non-leaders (or, *losers*). Note that each participant essentially proposes their *process index (PID)*, which, in a parameterized distributed system with an unbounded number of processes, is drawn from an infinite domain. To enable decidability of PMCP for systems that use partition-and-move agreement protocols, the cardinality of exactly one group must be unbounded (e.g., non-leaders), while that of all other groups must be finite (e.g., leaders). This disallows, for instance, partitioning the participants into two equal sets of winners and losers. Fortunately, we observe that most partition-and-move agreement protocols pick a finite number of winners that is independent of the number of participants.

Each **`Partition`** agreement primitive has an identifier, and takes two parameters: the set of participants and the desired number of winners. The reaction of each **`Partition`** primitive contains a **`win`** (resp. **`lose`**) handler that indicates how the process behaves upon winning (resp. losing).

Example. Distributed Store (Fig. 1) uses partition-and-move agreement in Line 13, modeled using a handler in the `Candidate` location with the agreement event: **`Partition`** `<elect>(All,1)`. This event uses the **`Partition`** primitive with identifier `elect` and is used to pick 1 process out of the set of all processes (`All`). The winners (resp. losers) of this agreement instance move to location `Leader` (resp. `Replica`).

The* `Consensus` *Primitive. The **`Consensus`** agreement primitive is used to model value-consensus agreement, where a set of participants, each proposing one value, wishes to decide on (a set of) values. Instances of value-consensus agreement include protocols such as Paxos [Lamport 1998], Fast Paxos [Lamport 2006], and Mencius [Mao et al. 2008]. To enable decidability of PMCP for systems that use value-consensus agreement, we restrict the domain of the proposed values to be finite. We note that many distributed systems aim to solve coordination-like problems rather than compute a function over their data. Hence, infinite concrete data domains can soundly be treated as finite abstract data domains using, for example, predicate abstraction.

Each **`Consensus`** primitive has an identifier, and takes three parameters: the set of participants, the number of proposals to be decided and (an optional) variable that a process uses to propose a value.

Example. Distributed Store (Fig. 1) uses value-consensus agreement in Lines 27 and 39 to allow the processes to decide which operation should be executed next. In particular, the processes use two handlers: one in the `RepCmd` location with the agreement event **`Consensus`** `<vcCmd>(All,1,cmd)`, and another in the `Replica` location with the agreement event **`Consensus`** `<vcCmd>(All,1,_)`. Both primitives have the identifier `vcCmd` and allow all processes to participate (`All`). In the former, the leader proposes a value from the variable `cmd`, while in the latter, the replicas propose no value (denoted by _). The primitive decides on one value that can be accessed by all participants using the expression `vcCmd.`**`decVar`**`[1]`.

Composition of Primitives. The **`Partition`** and **`Consensus`** primitives can be composed to model agreement protocols like Multi-Paxos [Chandra et al. 2007] and Raft [Ongaro and Ousterhout 2014], where a set of participants wish to decide on a potentially infinite sequence of values. Instead of invoking agreement on every value of the sequence individually, such protocols enhance practicality by first electing a leader that proposes the values, while the rest of the processes accept such values. Such protocols can be modeled by using a **`Partition`** primitive to elect a leader, and then using **`Consensus`** primitives to have the leader propose values in subsequent rounds. Our Distributed Store example uses such a composition: upon receiving a `doCmd` request, the leader uses the **`Consensus`** primitive to agree with the replicas. Note that the replicas pass an empty proposal (denoted _) as only the leader should be proposing values.

2.3 Parameterized Verification in QuickSilver

Since PMCP is a well-known undecidable problem, it is not immediately obvious if parameterized verification is even decidable for Mercury programs with agreement primitives. To this end, we first identify an expanded decidable fragment, Mercury Core, of an existing abstract model of distributed systems. Furthermore, we present two additional theoretical results that enable decidable and efficient parameterized verification for Mercury programs. These results are based on establishing a correspondence between Mercury programs and programs in the more abstract Mercury Core model, and appealing to Mercury Core's decidability and cutoff results.

Decidable Parameterized Verification. We identify syntactic conditions, called *phase-compatibility conditions*, on systems with agreement primitives that yield decidability of PMCP. Informally, the phase-compatibility conditions capture systems that proceed in *phases*: each process is always in the same phase as every other process and all processes, simultaneously, move from one phase to the next using some global synchronization such as synchronous broadcasts or agreement. The phase-compatibility conditions ensure that the system's ability to move between phases is independent of the number of processes, thereby paving the way towards decidable parameterized verification.

Example. Distributed Store (Fig. 1) is phase-compatible and hence enables decidable parameterized verification. The system starts in a phase where all processes are in the `Candidate` location, then uses a **`Partition`** primitive to move to the second phase where all processes are in locations `Leader` and `Replica` where the system is ready to receive requests from the clients.

Practical Parameterized Verification. Unfortunately, the decision procedure for PMCP in the Mercury Core fragment has non-primitive recursive complexity [Schmitz and Schnoebelen 2013]. Hence, we identify additional syntactic conditions, called *cutoff-amenability conditions*, that, for a given class of safety properties, enable reducing the parameterized verification problem for systems with phase-compatible processes to verification of a system with a small, fixed number of processes. This small, fixed number of processes is called a cutoff, and essentially entails a small model property: if there exists a counterexample to a safety property in a system with a certain, possibly large, number of processes, then there exists a counterexample to the property in a system with a cutoff number of processes.
Example. The cutoff for Distributed Store (Fig. 1) and its safety properties is 3. Essentially, due to the nature of the safety properties and the structure of the system, any violation of the property in a system with more than 3 actors can still be reproduced in a system with 3 process—any additional actors will not prevent the 3 process from potentially reaching an error state.

While the cutoff may seem obvious here, in general, cutoff arguments are non-trivial and deriving cutoff results requires a deep understanding of the underlying machinery for decidable parameterized verification.

We note that some Mercury programs may not be practically verifiable. In this event, QuickSilver provides best-effort feedback suggesting program modifications to the designer that can help *fit* their design into the desirable fragment of Mercury programs.

3 THE MERCURY MODELING LANGUAGE

We present Mercury, a language for modeling distributed agreement-based systems. We focus on systems in which the uncertainties and intricacies of their behavior in the presence of asynchronous communication and failures are essentially encapsulated within the underlying agreement protocols. Thus, while the agreement protocols may use asynchronous communication and tolerate network and process failures, we impose some simplifying assumptions on the system model *outside of* agreement. We assume that non-communicating processes can operate asynchronously, but communication is synchronous (i.e., sending and receiving processes must block until they can communicate). We further assume that processes may crash (i.e. exhibit crash-stop failures), but the network is reliable. These assumptions enable our initial exploration of the boundaries of decidable parameterized reasoning for agreement-based systems.

While Mercury includes standard features like communication actions, events, and event handlers, its distinguishing feature is the availability of *special primitives for encapsulating different agreement protocols*. Mercury also includes some design choices to facilitate decidable parameterized verification. We define the syntax and semantics of Mercury programs, with a detailed treatment of the semantics of its agreement primitives.

3.1 Mercury Syntax and Informal Semantics

Programs. A Mercury program is a collection of an unbounded number n of *identical*[2] *system processes* $P_1, P_2, \ldots, P_n$ and an *environment process* E, communicating via events. Each system process has a unique process index (PID) drawn from the set $I_n = \{1, 2, \ldots, n\}$.

[2]This can be relaxed to allow a finite number of distinct process definitions.

Processes. The syntax for a MERCURY system process is shown in Fig. 2. A process definition begins with a declaration of typed **`variables`** and communication **`actions`**, and is followed by a sequence of locations wit aa designated **`initial`** location. The variable type **`idSet`** corresponds to sets of process indices; the domain of this type is unbounded as the number of system processes is, in general, unbounded. The variable type **`int`** has a fixed, finite range that is specified in the declaration—this is one of the inbuilt restrictions in MERCURY to facilitate automated parameterized verification. Communication actions either represent communication between system processes or communication between the environment process and system processes; further, communication actions are either broadcast actions (denoted **`br`**) involving communication from one process to all other processes or rendezvous actions (denoted **`rz`**) involving communication between a pair of processes. Each communication action `act` has an optional (finite) integer-valued *payload* field that can be retrieved via the expression `act.`**`payld`**.

Each location contains a set of event handlers that consists of an event and a reaction to that event. An event can be the empty event (_), a receive of a communication action (**`recv`**), or one of two agreement primitives. A handler for an empty event corresponds to a *non-reactive* action a process may initiate, i.e., an internal computation or a send of a communication action. A **`Partition`** primitive `part` has two parameters: the set of participants and the number of *winners* to be chosen. The set of winners (resp. *losers*) is retrieved via the expression `part.`**`winS`** (resp. `part.`**`loseS`**). A **`Consensus`** primitive `cons` has three parameters: the set of participants, the number of proposals to be chosen, and an optional variable, ⟨*optIntVar*⟩, from which a process proposes its value. The k^{th} value from the set of decided values, `cons.`**`decVar`**, is retrieved via the expression `cons.`**`decVar[`**k**`]`**.

A reaction to an event consists of a block of update statements, control statements, and/or sends. Update statements include assignments to integer variables and statements to add or remove a PID from a set of PIDs. Control statements include conditionals and **`goto`** statements used to switch between locations. A **`sendrz`** statement is a rendezvous send that transmits a message with an action identifier `act` and an optional payload ⟨*optIntVar*⟩ to a process with index given by ⟨*idExp*⟩. A **`sendbr`** statement is a broadcast send that transmits a message with an action identifier `act` and an optional payload ⟨*optIntVar*⟩ to all other processes.

Reactions for empty, receive and **`Consensus`** events all begin with **`do`**. Additionally, a *guarded reaction* of the form **`where(`**⟨*bExp*⟩**`) do`** ⟨*stmt*⟩ can be used for empty and receive events to ensure that the handler is only enabled if some Boolean predicate evaluates to **`true`**. Finally, the reaction for a **`Partition`** event is of the form **`win:`** ⟨*stmt*⟩ **`lose:`** ⟨*stmt*⟩, indicating how a process should react if it wins and if it loses.

An environment process is a simpler version of a system process with variable types restricted to **`int`** and event types restricted to empty and receive events.

Expressions. The syntax for all expressions in MERCURY processes is shown in Fig. 3. An ⟨*idExp*⟩ expression evaluates to a PID—**`self`** retrieves the PID of the current process and, in a receive handler for communication action `act`, the expression `act.`**`sID`** retrieves the PID of the corresponding sender. An ⟨*idSetExp*⟩ expression evaluates to a set of PIDs as shown and includes the expressions `part.`**`winS`** and `part.`**`loseS`** introduced earlier. An ⟨*intExp*⟩ expression evaluates to an integer and includes the expressions `act.`**`payld`** and `cons.`**`decVar[`**`intConst`**`]`**, introduced earlier. A MERCURY arithmetic expression, ⟨*arithOp*⟩, is standard and is not shown. A MERCURY Boolean expression, ⟨*bExp*⟩, constrains comparison of ⟨*idExp*⟩ expressions to equality and disequality checks; we expand on this restriction at the end of Sec. 3.2 and emphasize that this is a common syntactic restriction used to facilitate the use of *structural symmetries* for scalable verification (cf. [Emerson and Sistla 1996; Emerson and Wahl 2003; Ip and Dill 1996; v. Gleissenthall et al. 2019; Wahl 2007]).

```
⟨process⟩     ::= process proc ; variables ⟨vars⟩ ;  actions ⟨acts⟩ ; initial ⟨locs⟩
⟨vars⟩        ::= ϵ | ⟨vars⟩ ; ⟨vars⟩
                | idSet idSetV                                  Set of PIDs initialized to Empty
                | int [intConst, intConst] intV := intConst     User-initialized bounded integer
⟨acts⟩        ::= ⟨sysActs⟩ ; ⟨envActs⟩
⟨sysActs⟩     ::= ϵ | ⟨sysActs⟩ ; ⟨sysActs⟩
                | br act : ⟨payldDom ⟩                          Broadcast action
                | rz act : ⟨payldDom ⟩                          Rendezvous action
⟨envActs⟩     ::= ϵ | env ⟨sysActs⟩
⟨payldDom⟩    ::= unit                                        Empty payload
                | int [intConst, intConst]                     Bounded integer payload
-------------------------------------------------------------------------------
⟨locs⟩        ::= ⟨locs⟩ ; ⟨locs⟩
                | location loc ⟨handlers⟩
⟨handlers⟩    ::= ϵ | ⟨handlers⟩ ; ⟨handlers⟩
                | on ⟨event⟩ ⟨reaction⟩
⟨event⟩       ::= _                                           Empty event
                | recv(act)                                    Receive event
                | Partition<part>(⟨idSetExp⟩, intConst)        Partition event
                | Consensus <cons>(⟨idSetExp⟩, intConst, ⟨optIntVar⟩)   Consensus event
⟨reaction⟩    ::= do ⟨stmt⟩                                   Unguarded reaction
                | where(⟨bExp⟩) do ⟨stmt⟩                      Guarded reaction
                | win: ⟨stmt⟩ lose: ⟨stmt⟩                     Partition reaction
⟨stmt⟩        ::= ⟨stmt⟩ ; ⟨stmt⟩
                | ⟨updateStmt⟩
                | ⟨sendStmt ⟩
                | ⟨controlStmt⟩
⟨updateStmt⟩  ::= intV := ⟨intExp⟩
                | idSetV.add(⟨idExp⟩)
                | idSetV.remove(⟨idExp⟩)
⟨sendStmt⟩    ::= sendrz(act, ⟨optIntVar⟩, ⟨idExp⟩)           Rendezvous send statement
                | sendbr(act, ⟨optIntVar⟩)                     Broadcast send statement
⟨controlStmt⟩ ::= if(⟨bExp⟩) ⟨stmt⟩ else ⟨stmt⟩
                | goto loc
⟨optIntVar⟩   ::= _ | intV                                     Optional integer variable
```

Fig. 2. Syntax for Mercury. In the grammar, non-terminals are enclosed in ⟨ ⟩, keywords are in **boldface**, and all other terminals are monospaced.

Syntactic Sugar. Mercury provides syntactic sugar (Fig. 4) to simplify expressing some common idioms. A **passive** handler specifies events a process should *not* react to. A **reply** reaction sends a rendezvous reply to the last sender.

3.2 Agreement-Free Mercury Program Semantics

The semantics of Mercury processes and programs is best described using state-transition systems. We first define the semantics of Mercury programs without agreement primitives, then extend the definition to Mercury programs with agreement primitives in Sec. 3.3. Intuitively, the semantics

⟨*idExp*⟩ ::= **self** — PID of current process
| act.**sID** — PID of sender of action act

⟨*idSetExp*⟩ ::= **All** | **Empty** | idSetV
| part.**winS** — Set of winners of **Partition** primitive part
| part.**loseS** — Set of losers of **Partition** primitive part

⟨*intExp*⟩ ::= intConst | intV | ⟨*intExp*⟩ ⟨*arithOp*⟩ ⟨*intExp*⟩
| act.**payld** — Payload of action act
| cons.**decVar[**intConst**]** — Selecting some decided value of **Consensus** primitive cons

⟨*bExp*⟩ ::= **True** | **False** | **!**⟨*bExp*⟩
| ⟨*bExp*⟩ ⟨*boolOp*⟩ ⟨*bExp*⟩
| ⟨*intExp*⟩ ⟨*cmpOp*⟩ ⟨*intExp*⟩
| ⟨*idExp*⟩ ⟨*eqOp*⟩ ⟨*idExp*⟩

⟨*cmpOp*⟩ ::= **<** | **>** | **<=** | **>=** | ⟨*eqOp*⟩

⟨*eqOp*⟩ ::= **=** | **!=**

Fig. 3. Syntax of Mercury Expressions.

⟨*handlers*⟩ ::= **passive** ⟨*eventList*⟩ — Handler specifying list of events a process should *not* react to

⟨*eventList*⟩ ::= ⟨*eventId*⟩ | ⟨*eventId*⟩ **,** ⟨*eventList*⟩

⟨*eventId*⟩ ::= act | part | cons

⟨*reaction*⟩ ::= **reply(**act**,** ⟨*optIntVar*⟩**)** — A rendezvous *reply* to the last sender

Fig. 4. Mercury Syntactic Sugar.

allows non-communicating processes in Mercury programs to operate asynchronously while ensuring that communication and agreement is synchronous and consistent.

Core Fragment of Mercury. To enable a succinct description of Mercury programs' semantics, we rewrite process definitions into a core fragment of the language with the event handlers and statements depicted in Fig. 5. The handlers may contain two types of statements (shown in Fig. 5a): a statement ⟨*iStmt*⟩ that consists of a (possibly empty) sequence of update statements, followed by a **goto** statement; and a statement ⟨*sStmt*⟩ that consists of a send statement, followed by an ⟨*iStmt*⟩ statement. The *internal* core handler in Fig. 5b embodies computations that the process does without communication with other processes; the *send* core handler in Fig. 5c embodies a send of some action by a process; and the *receive* core handler in Fig. 5d embodies the reaction of a process to a receive of some action. Note that all three handlers are guarded by a predicate that dictates when they are enabled. The core handlers in Fig. 5e and Fig. 5f are for agreement primitives and only contain **goto** statements as shown. For the rest of this paper, let P be a process in the core frament of Mercury. With some abuse of notation, we use *vars*, *acts*, and *locs* to refer to the sets of variables, actions, and locations in their eponymous sequences in Fig. 2.

Process Semantics. The semantics of a process P is defined as a labeled state-transition system $(S, s_0, s_{cr}, acts, T)$, where S is the set of (local) states, s_0 is the initial state, s_{cr} is a special "crashed" state, *acts* is the set of actions, and $T \subseteq S \times \{\texttt{sendrz}, \texttt{sendbr}, \texttt{recvrz}, \texttt{recvbr}, \texttt{crash}, \epsilon\} \times acts \times I_n \times S$ is the set of (local) labeled transitions of P. A state $s \in S$ is a pair $(\texttt{loc}, \sigma)$ where $\texttt{loc} \in locs$ is a

⟨*iStmt*⟩ ::= ⟨*uStmts*⟩; **goto** loc

⟨*sStmt*⟩ ::= ⟨*sendStmt* ⟩; ⟨*iStmt*⟩

⟨*uStmts*⟩ ::= ϵ | ⟨*updateStmt*⟩ | ⟨*uStmts*⟩; ⟨*uStmts*⟩ (a)

```
location loc
 on _ where (⟨bExp⟩) do
  ⟨iStmt⟩
```
(b)

```
location loc
 on _ where (⟨bExp⟩) do
  ⟨sStmt⟩
```
(c)

```
location loc
 on recv(act) where (⟨bExp⟩) do
  ⟨iStmt⟩
```
(d)

```
location loc
 on Partition<part>(⟨idSetExp⟩,intConst)
  win: goto loc lose: goto loc
```
(e)

```
location loc
 on Consensus<cons>(⟨idSetExp⟩,intConst,⟨optIntVar⟩)
   goto loc
```
(f)

Fig. 5. Syntax of Core Mercury. (a) Core Statements. Core handler for (b) Internal, (c) Send, (d) Receive, (e) **Partition**, and (f) **Consensus**.

location and σ is a valuation of the variables in *vars*. We let $\sigma(var)$ denote the value of the variable *var* according to σ. For a state $s = (\texttt{loc}, \sigma)$, we let $s.loc$ denote the location loc in s, and $s.\sigma(var)$ denote the value $\sigma(var)$ of variable *var* in s. Similarly, we use $\sigma(expr)$ ($s.\sigma(expr)$) to denote the value of expression *expr* evaluated under σ (in state s). The initial state $s_0 = (\texttt{loc}_0, \sigma_0)$, where $\texttt{loc}_0$ denotes the initial location and σ_0 denotes the initial variable valuation. The crash state s_{cr} is a special state that the process is assumed to enter upon exhibiting a crash-stop failure.

A transition of process P without agreement primitives corresponds to the execution of one of the three core event handlers in Fig. 5b, Fig. 5c, and Fig. 5d; a transition is labeled either with a send/receive of a communication action in *acts* or an empty label ϵ denoting an internal transition. For each core handler, let loc denote the current location and loc′ denote the target location of the **goto** statement. Then, the transitions in T are defined as follows:

(a) For each broadcast send handler as shown in Fig. 5c with ⟨*sendStmt*⟩ given by **sendbr**(act, ⟨*optIntVar*⟩), T contains a transition $(\texttt{loc}, \sigma) \xrightarrow{\textbf{sendbr}(\texttt{act})} (\texttt{loc}', \sigma')$ for each σ such that $\sigma(bExp) = true$ and σ' is obtained from σ by applying the sequence of updates ⟨*uStmts*⟩. Note that if ⟨*optIntVar*⟩ is ϵ, the payload is empty and if ⟨*optIntVar*⟩ is a variable, denoted by $var_{\texttt{act}}$, the payload is $\sigma(var_{\texttt{act}})$.

(b) For each broadcast receive handler as shown in Fig. 5d with broadcast action act, T contains a transition $(\texttt{loc}, \sigma) \xrightarrow{\textbf{recvbr}(\texttt{act})} (\texttt{loc}', \sigma')$ for each σ such that $\sigma(bExp) = true$ and σ' is obtained from σ by applying the sequence of updates ⟨*uStmts*⟩. Note that ⟨*uStmts*⟩ may access the received value using the expression act.**payld**.

(c) To model process crash-stop failures, T contains a transition $(\texttt{loc}, \sigma) \xrightarrow{\texttt{crash}} s_{cr}$ for each $(\texttt{loc}, \sigma)$.

Local transitions corresponding to internal and rendezvous send and receive can be formalized similarly. We use E to denote the environment process and S_E, $s_{0,E}$ etc. to denote its set of states, initial state etc., respectively.

Distributed Program Semantics. The semantics of a MERCURY program consisting of n identical system processes $P_1, \ldots, P_n$ and the environment process E is defined as a state-transition system $\mathcal{M}(n) = (Q, q_0, R)$, parameterized by the number of processes n, where:

1. $Q = S^n \times S_E$ is the set of global states,
2. $q_0 = (s_0, \ldots, s_0, s_{0,E})$ is the initial global state, and
3. $R \subseteq Q \times Q$ is the set of global transitions where (i) all processes synchronize on a broadcast communication action, (ii) two processes synchronize on a rendezvous communication action, (iii) one process makes an asynchronous internal move, or (iv) one process crashes. Formally, a global transition (q, q') based on a broadcast action `act` is in R iff there exists a process P_i with $s_i \xrightarrow{\texttt{sendbr(act)}} s_i'$, and every other process P_j with $j \neq i$ has a transition $s_j \xrightarrow{\texttt{recvbr(act)}} s_j'$ such that $q' = q[s_i \leftarrow s_i', \forall j \neq i : s_j \leftarrow s_j']$ and if $\langle optIntVar \rangle$ is the variable $var_{\texttt{act}}$, $s_i.\sigma(var_{\texttt{act}}) = \texttt{act.}\textbf{payld}$. Here, $q[s_i \leftarrow s_i']$ indicates that process P_i moves from state s_i to s_i'. A global crash transition (q, q') is in R iff there exists a process P_i with a local crash transition $s_i \xrightarrow{\text{crash}} s_{cr}$ and $q' = q[s_i \leftarrow s_{cr}]$. Global transitions corresponding to rendezvous actions or internal transitions can be formalized similarly.

An execution of a global transition system $\mathcal{M}(n)$ is a (possibly infinite) sequence of states, $q_0, q_1, \ldots$, in Q such that for each $j \geq 0$, $(q_j, q_{j+1}) \in R$. A state q is *reachable* if there exists a finite execution of $\mathcal{M}(n)$ that ends in q.

In what follows, we use $\mathcal{M}$ and $\mathcal{M}(n)$ as well as system process and process, interchangeably.

Correctness Specifications. In this work, we focus on a broad class of invariant properties of systems modeled in MERCURY. In particular, our correctness specifications are Boolean combinations of universally quantified formulas over locations, **`int`** variables, and a finite number of variables with distinct valuations over I_n.

For example, one can specify that a location c is a critical section (of size 1) as: $\forall i, j \in I_n. \neg(q[i].loc = c \wedge q[j].loc = c)$; Distributed Store (Fig. 1) uses a specification of this form to ensure that at most 1 process can be in `Leader`. As another example, one can specify that all processes in some location d must have the same value in their local variable v as: $\forall i, j \in I_n. q[i].loc = d \wedge q[j].loc = d \implies q[i].\sigma(v) = q[j].\sigma(v)$; Distributed Store uses specifications of this form to ensure the stored data is consistent.

The program $\mathcal{M}(n)$ is *safe* if it has no reachable states that violate its correctness specification. Given a specification $\phi(n)$, also parameterized by n, we use the standard notation $\mathcal{M}(n) \models \phi(n)$ to denote that $\mathcal{M}(n)$ is safe.

Symmetry for Efficient, Parameterized Verification. Performing automated parameterized verification for systems with an arbitrary number of processes hinges on the number of different *types* of processes being *bounded* (in MERCURY, there are two types: system and environment). Thus, parameterized systems naturally exhibit many *similar* global behaviors that are independent of specific process indices. The symmetric nature of such global behaviors offers another advantage: it is possible to greatly improve the verification time of symmetric systems through *symmetry reduction* [Emerson and Sistla 1996]. In particular, a (global) state-transition system $\mathcal{M}$ is *fully-symmetric* if its transition relation R is invariant under permutations over the set I_n of PIDs. As noted in Sec. 3.1, MERCURY syntactically constrains comparison of $\langle idExp \rangle$ expressions to (dis)equality checks. This is a sufficient condition to ensure MERCURY processes are fully-symmetric and, hence, enable parameterized verification and symmetry reduction.

3.3 Semantics of Mercury Agreement Primitives

We now extend the process and program semantics defined in Sec. 3.2 to Mercury programs with agreement primitives. Furthermore, we show that our definition of the semantics of agreement primitives provides a sound abstraction of agreement protocols and enables symmetry reduction.

To simplify the presentation of the semantics, we expand the set *vars* of variables as follows. For each **Partition** event `part`, we add variables `part_winS` and `part_loseS` for storing the sets of winners and losers, respectively. Similarly, for each **Consensus** event `cons`, we add variable `cons_decVar` for storing the decided values.

Process-level Semantics of Agreement Primitives. For the **Partition** event handler (Fig. 5e), let `loc` be the current location and $\texttt{loc}_w$ (resp. $\texttt{loc}_l$) be the target location of the `goto` statement in the **win:** (resp. **lose:**) block. For the **Consensus** event handler (Fig. 5f), let `loc` denote the current location and $\texttt{loc}_d$ denote the target location of the `goto` statement. Then, the set T of transitions is extended as follows:

(a) For each **Partition** handler with event **Partition** <part> (⟨*idSetExp*⟩,`intConst`) in Fig. 5e, T contains transitions $(\texttt{loc}, \sigma) \xrightarrow{\texttt{win}:PC_{\texttt{part}}(\texttt{pcpt},k)} (\texttt{loc}_w, \sigma')$ and $(\texttt{loc}, \sigma) \xrightarrow{\texttt{lose}:PC_{\texttt{part}}(\texttt{pcpt},k)} (\texttt{loc}_l, \sigma')$ for each σ such that `pcpt`, matched by ⟨*idSetExp*⟩, denotes the set of participants[3], k, given by `intConst`, denotes the number of winners to be decided, and σ' is obtained from σ by updating variables `part_WinS` and `part_loseS` to the sets of winners and losers in the global invocation of `part`, respectively.

(b) For each **Consensus** handler with event **Consensus** <cons> (⟨*idSetExp*⟩, `intConst`, ⟨*optIntVar*⟩) in Fig. 5f, T contains a local transition $(\texttt{loc}, \sigma) \xrightarrow{VC_{\texttt{cons}}(\texttt{pcpt},k,\texttt{pVar})} (\texttt{loc}_d, \sigma')$ for each σ such that `pcpt` and k are as before, `pVar` denotes the variable from which a process proposes its value, matched by ⟨*optIntVar*⟩ if ⟨*optIntVar*⟩ is not ϵ, and σ' is obtained from σ by updating the variable `cons_decVar` to the decided values in the invocation of `cons`.

Program-level Semantics of Agreement Primitives. The local transitions corresponding to agreement primitives are essentially *modeling* invocation of verified agreement protocols that enable a set of participants to decide on a finite set of winners/values in a *globally consistent* way. As stated in Sec. 2, verified agreement protocols typically entail agreement, validity, and termination. Thus, to ensure that agreement primitives provide a sound abstraction of verified agreement protocols, the *global behavior* of these primitives must satisfy a set of conditions entailed by agreement, validity, and termination. We represent this set of conditions on the global transitions corresponding to agreement primitives as a precondition-postcondition pair, stated informally as:

C_1: *Consistent Participants Precondition.* The participants agree on *with whom* to invoke agreement [4], and,

C_2: *Consistent Decisions Postcondition.* Upon termination of agreement, all *non-crashed* participants concur on winners/values.

In what follows, we present the global transitions and specialization of the precondition-postcondition pair (C1,C2) for each type of agreement primitive.

Partition. Consider an instance of a **Partition** agreement primitive with identifier `part` and local transitions $(\texttt{loc}^c, \sigma) \xrightarrow{\texttt{win}:PC_{\texttt{part}}(\texttt{pcpt},k)} (\texttt{loc}^c_w, \sigma')$ and $(\texttt{loc}^c, \sigma) \xrightarrow{\texttt{lose}:PC_{\texttt{part}}(\texttt{pcpt},k)} (\texttt{loc}^c_l, \sigma')$.

[3]While such sets are usually predefined, we allow more flexibility by permitting processes to communicate and construct them.

[4]Systems in which *all* processes intend to reach agreement trivially satisfy C_1. The more general form of C_1 enables systems to invoke agreement protocols with only a subset of processes.

Let $locs^{\texttt{part}}$ be the set of all locations $\texttt{loc}^c$ from which the participants of this instance may invoke **Partition** (i.e., all locations where the above two transitions originate).

We extend the global transition relation R of $\mathcal{M}$ with a **Partition** agreement transition from global state $q_{\texttt{start}}$ to global state $q^W_{\texttt{end}}$ encoding a selected set W of k[5] non-crashed winners, and a set F of participants that have crashed during agreement if:

$C_1(PC)$: There exists a set $S \subseteq I_n$ of processes in $q_{\texttt{start}}$ in appropriate locations for invoking this instance of the **Partition** primitive and with a consistent view of each other. Formally:
(1) $\forall i \in S : q_{\texttt{start}}[i].loc \in locs^{\texttt{part}}$ and
(2) $\forall i, j \in S : q_{\texttt{start}}[i].\sigma(\texttt{pcpt}) = q_{\texttt{start}}[j].\sigma(\texttt{pcpt}) = S$, and,

$C_2(PC)$: The non-crashed processes of S move to their appropriate target locations in $q^W_{\texttt{end}}$ based on whether they *win* or *lose* and their `part_winS` and `part_LoseS` variables in $q^W_{\texttt{end}}$ are updated to reflect the partition while the set $F \subset S$ of crashed processes move to the crash state s_{cr}. As explained in Remark 1 below, we assume that if all the participants fail, then no valid $q^W_{\texttt{end}}$ exists. Formally: Let N be the set $S \setminus F$ of non-crashed participants, then:
(1) $\forall i \in N : i \in W \wedge q_{\texttt{start}}[i].loc = \texttt{loc}^c \Rightarrow q^W_{\texttt{end}}[i].loc = \texttt{loc}^c_w$,
(2) $\forall i \in N : i \notin W \wedge q_{\texttt{start}}[i].loc = \texttt{loc}^c \Rightarrow q^W_{\texttt{end}}[i].loc = \texttt{loc}^c_l$,
(3) $\forall i \in N : q^W_{\texttt{end}}[i].\sigma(\texttt{part\_winS}) = W$,
(4) $\forall i \in N : q^W_{\texttt{end}}[i].\sigma(\texttt{part\_LoseS}) = N \setminus W$,
(5) $\forall i \in F : q^W_{\texttt{end}}[i] = s_{cr}$, and,
(6) $\forall i \in I_n \setminus S : q^W_{\texttt{end}}[i] = q_{\texttt{start}}[i]$.

Consensus. Consider an instance of a **Consensus** agreement primitive with identifier `cons` and local transition $(\texttt{loc}^c, \sigma) \xrightarrow{VC_{\texttt{cons}}(\texttt{pcpt},k,\texttt{pVar})} (\texttt{loc}^c_d, \sigma')$. As before, let $locs^{\texttt{cons}}$ be the set of locations $\texttt{loc}^c$ from which the participants of this `cons` instance may start.

We extend the global transition relation R of $\mathcal{M}$ with a **Consensus** agreement transition from a global state $q_{\texttt{start}}$ to a global state $q^W_{\texttt{end}}$ encoding a selected set W of k decided values[6] and a set F of participants that have crashed during agreement if:

$C_1(VC)$: The state $q_{\texttt{start}}$ is as defined for **Partition**, and,

$C_2(VC)$: The set $N \coloneqq S \setminus F$ such that $|N| > |F|$ of non-crashed processes move to their target locations in $q^W_{\texttt{end}}$ and their `cons_decVar` variables are updated to reflect the decided values while the set F of crashed processes move to the crash state s_{cr}. As explained in Remark 1 below, we assume that if a majority of participants fail (i.e., $|N| \leq |F|$), then no valid $q^W_{\texttt{end}}$ exists. Formally:
(1) $\forall i \in N : q_{\texttt{start}}[i].loc = \texttt{loc}^c \Rightarrow q^W_{\texttt{end}}[i].loc = \texttt{loc}^c_d$,
(2) $\forall i \in N : \; q^W_{\texttt{end}}[i].\sigma(\texttt{cons\_decVar}) = W$,
(3) $\forall i \in F : q^W_{\texttt{end}}[i] = s_{cr}$, and,
(4) $\forall i \in I_n \setminus S : q^W_{\texttt{end}}[i] = q_{\texttt{start}}[i]$.

Remark 1: Failure Assumptions for Valid Termination. Common agreement protocols have assumptions about process failures under which they guarantee the validity of results upon termination. For instance, leader election protocols [Garcia-Molina 1982] require the elected leader to not fail, but can tolerate the failures of the losing processes and consensus protocols like Paxos [Lamport 1998] and Raft [Ongaro and Ousterhout 2014] require a simple majority of the participants to not fail and to agree on a proposed value. While, in general, the semantics of Mercury's agreement primitives can be parameterized over specific failure assumptions, our default definitions encode

[5] All non-crashed participants act as winners if the number of non-crashed participants is less than k.
[6] Note that, a proposed value of a crashed process can still be chosen as the decided value.

these common assumptions. Thus, when using the **`Partition`** primitive, any global state q^W_{end} where all the participants have failed is assumed to be not valid. When using the **`Consensus`** primitive, any global state q^W_{end} where a majority of the participants have failed is assumed to be not valid.

Soundness. Mercury's agreement primitives are sound:

Lemma 3.1. *Our proposed abstraction of verified agreement protocols, as defined using the syntax and semantics of Mercury agreement primitives, is sound. In other words, if an agreement protocol satisfies agreement, validity, and termination, then the agreement protocol satisfies the semantics of agreement primitives captured by the precondition-postcondition pair* (C_1, C_2).

Proof. We prove this by contradiction. Assume that an agreement protocol satisfying agreement, validity, and termination begins in a state $q_{\texttt{start}}$ that satisfies precondition C_1[7] but ends in a state q^W_{end} that violates postcondition C_2. A violation of postcondition C_2 (i.e., participants not agreeing on the same winner/value or agreeing on a winner/value that was not in the set of participants/was never proposed) contradicts agreement and validity. Finally, a violation due to the absence of a transition between a state $q_{\texttt{start}}$ satisfying C_1 and a state q^W_{end} satisfying C_2 directly contradicts termination.

Note that the statement of Lemma 3.1 implicitly assumes that the failure assumptions encoded in the semantics of Mercury agreement primitives hold. If the failure assumptions do not hold, neither our primitives nor the agreement protocols they abstract provide any guarantees.

Symmetry. In a state-transition system, $\mathcal{M}_{\text{AGREE}} = (Q, q_0, R)$, capturing the semantics of a Mercury program, let R_{AGREE} be the set of all transitions corresponding to agreement primitives in R. Let $\mathcal{M} = (Q, q_0, R \setminus R_{\text{AGREE}})$ be the state-transition system without the *agreement transitions* of $\mathcal{M}_{\text{AGREE}}$.[8]

Lemma 3.2. *If* $\mathcal{M}$ *is fully-symmetric, then* $\mathcal{M}_{AGREE}$ *is fully-symmetric.*

Intuitively, the proof (ref. extended version [Jaber et al. 2020b]) is based on the observation that agreement transitions are oblivious to the identities of the participants and are hence invariant under permutations over I_n.

4 VERIFICATION OF MERCURY PROGRAMS

We now formalize the parameterized verification problem for Mercury programs and present our theoretical results for enabling decidable and efficient parameterized verification.

Mercury Parameterized Verification Problem (MPVP). Given a Mercury system process P, an environment process E, and a parameterized safety specification $\phi(n)$ as defined in Sec. 3, MPVP asks if $\forall n.\ \mathcal{M}(n) \models \phi(n)$.

Our first result (Sec. 4.1) identifies conditions on Mercury programs and the specification $\phi(n)$ for enabling decidability of MPVP. Our second result (Sec. 4.2) identifies additional conditions for which this problem is *efficiently* decidable, based on *cutoff* results. Cutoff results reduce the parameterized verification problem to a verification problem over a *fixed* number of processes. Formally, a cutoff for a parameterized system $\mathcal{M}$ and correctness specification ϕ is a number $c \in \mathbb{N}$ such that:

$$\forall n \geq c.\ (\mathcal{M}(c) \models \phi(c) \iff \mathcal{M}(n) \models \phi(n)).$$

[7] A violation of precondition C_1 (i.e., participants do not have a consistent view of each other or are in invalid local states to participate in the agreement protocol) indicates an invalid global state to invoke agreement.

[8] We do not make any symmetry-related assumptions about the specific agreement protocol that an agreement primitive encapsulates. In particular, the underlying agreement protocol could employ non-symmetric strategies such as "the process with maximum PID wins".

In particular, our second result identifies conditions on Mercury programs for *small* cutoffs, reducing MPVP to verification of a Mercury program with a *small* number of processes. The latter problem is decidable for any Mercury program, as the corresponding semantics can be expressed as a finite-state machine.

For the rest of this section, we fix a Mercury process P with a set of process-local states S, initial state s_0, and process-local transitions T, and refer to the corresponding global state-transition system, $\mathcal{M}$, as a (Mercury) system.

4.1 Decidable Parameterized Verification

Phases. To enable decidable parameterized verification, we view Mercury systems as proceeding in *phases*. A phase of a Mercury system is a set of process-local states, characterizing the set of events that can occur when *all* processes co-exist in that set of local states. In any global execution, all processes simultaneously move from one phase to the next, where a new set of events may occur. Processes move between phases strictly via *globally-synchronizing events*, i.e, broadcasts or agreement primitives; within a phase, processes can use any type of communication. While two phases may share some local states, their associated events are disjoint. We note that any Mercury system can be viewed as proceeding in phases, by identifying phases with appropriate sets of events—so phases do not constrain the applicability of our approach.

In what follows, we refer to the set of globally-synchronizing events as $E_{\texttt{global}}$ and the set of rendezvous actions in P as $E_{\texttt{rend}}$. For each event $\texttt{e}$, we define its source set, denoted $src_{\texttt{e}}$, as the set of states in S *from which* there exists a transition in T labeled with $\texttt{e}$. Similarly, we define the destination set of each event $\texttt{e}$, denoted $dst_{\texttt{e}}$, as the set of states in S *to which* a transition in T labeled with $\texttt{e}$ exists. For instance, if $\texttt{e}$ is a broadcast action $\texttt{act}$, $src_{\texttt{act}} = \{s \mid s \xrightarrow{\textbf{sendbr}(\texttt{act})} s' \in T \ \vee \ s \xrightarrow{\textbf{recvbr}(\texttt{act})} s' \in T\}$ and $dst_{\texttt{act}} = \{s' \mid s \xrightarrow{\textbf{sendbr}(\texttt{act})} s' \in T \ \vee \ s \xrightarrow{\textbf{recvbr}(\texttt{act})} s' \in T\}$. The source and destination sets for rendezvous actions and instances of **`Consensus`** and **`Partition`** can be defined similarly. Finally, we define the relation $\mathcal{R} \subseteq S \times S$ to denote pairs of states related via internal or rendezvous transitions as follows:

$$\mathcal{R} = \{(s,t) \mid s \neq t \ \wedge \ (s \xrightarrow{\epsilon} t \in T \ \vee \ t \xrightarrow{\epsilon} s \in T \ \vee \ (\exists \texttt{e} \in E_{\texttt{rend}}.\ \{s,t\} \subseteq src_{\texttt{e}} \cup dst_{\texttt{e}}))\}.$$

We now present a constructive definition for the set of phases of S. Intuitively, two states are in the same phase if they are part of the same source or destination set, or, their phases are *connected* by internal or rendezvous transitions.

Definition 4.1 (Phases). The set of phases is constructed as follows:

(1) *Initialization*: The set of phases is initialized to the set of source sets and destination sets of each globally-synchronizing event:

$$inPhases = \bigcup_{\texttt{e} \in E_{\texttt{global}}} \{src_{\texttt{e}},\ dst_{\texttt{e}}\}.$$

Informally, the source set of a globally-synchronizing event $\texttt{e}$ is a subset of the local state space where *all* processes need to co-exist for $\texttt{e}$ to occur. The destination set of $\texttt{e}$ characterizes the set of states in which *all* processes co-exist after event $\texttt{e}$ has occurred.

(2) *Expansion*: Each initial phase is then expanded such that if a state s is in a phase, then every state t such that $\mathcal{R}(s,t)$ holds is in the phase too:

$$exPhases = \bigcup_{X \in inPhases} \{X \cup \{t \mid s \in X \wedge \mathcal{R}^+(s,t)\}\},$$

where $\mathcal{R}^+$ is the transitive closure of $\mathcal{R}$. Informally, this step ensures that any local state that is reachable from or can be reached by a state in an initial phase via internal or rendezvous transitions, is added to that phase.

(3) *Merge*: Finally, expanded phases that contain distinct states s, t with $\mathcal{R}(s,t)$ are merged:

$$phases = \Big\{ \bigcup_{W \in X} W \mid X \subseteq exPhases \wedge \forall\, Y, Z \in X.\, \mathcal{R}^+_{ph}(Y,Z) \Big\},$$

where $\mathcal{R}_{ph} = \{(X,Y) \mid \exists s \in X, t \in Y.\, \mathcal{R}(s,t)\}$ and $\mathcal{R}^+_{ph}$ is the transitive closure of $\mathcal{R}_{ph}$. Informally, if processes can move via internal or rendezvous transitions between two expanded phases, then the two phases are merged to ensure that the processes always co-exist in the same phase.

This definition ensures that all the processes in the system are in the same phase at any given time in a program execution.

Phase-compatibility Conditions. Our *phase-compatibility* conditions ensure that all processes in a Mercury system move in phases such that the set of available events within a phase as well as the system's ability to move between phases (through globally-synchronizing events) is *independent of the number of processes*. Such independence is critical for decidability of MPVP, which needs to reason about an arbitrary number of processes. In particular, since processes can only move between phases using globally-synchronizing events, these conditions ensure that such events behave in a way that is independent of the number of processes. A process that satisfies these conditions is called *phase-compatible*. In what follows, we present the phase-compatibility conditions.

We first define a classification of local transitions corresponding to globally-synchronizing events into *acting* and *reacting* transitions. For broadcasts, sending transitions are acting while receiving transitions are reacting. For the **`Partition`** primitive, winning transitions are acting while losing transitions are reacting. For the **`Consensus`** primitive, transitions with winning proposals are acting while other transitions are reacting. For each globally-synchronizing event e, let $s \xrightarrow{A(\mathsf{e})} s'$ (resp. $s \xrightarrow{R(\mathsf{e})} s'$) denote a local acting (resp. reacting) transition of e. Additionally, for some event e and some subset X of the local state space S, we say that e is *initiable* in X if some state in X has an acting transition of e.

Definition 4.2 (Phase-Compatibility Conditions).

(1) Every state $s \in P$ which has an acting transition $s \xrightarrow{A(\mathsf{e})} s'$ must also have a corresponding reacting transition $s \xrightarrow{R(\mathsf{e})} s''$.

(2) For each internal transition $s \rightarrow s'$ that is accompanied by a reacting transition $s' \xrightarrow{R(\mathsf{f})} s''$ and for each state t in the same phase as s, if event f is initiable in that phase, then t must have a path to a state with a reacting transition of event f.

(3) For each acting transition $s \xrightarrow{A(\mathsf{e})} s'$ that is accompanied by a reacting transition $s' \xrightarrow{R(\mathsf{f})} s''$ such that f is initiable in the set dst_{e} of *destination states of event* e, (i) if there are other acting transitions $t \xrightarrow{A(\mathsf{e})} t'$ for event e, all of them must transition to a state t' with a reacting transition $t' \xrightarrow{R(\mathsf{f})} t''$ of event f and (ii) for every reacting transition $u \xrightarrow{R(\mathsf{e})} u'$ of e, there must be a path from u' to a state with a reacting transition of event f.

Intuitively, the conditions ensure that if a Mercury system with a given number of processes can move in phases, then any additional processes can go along with the existing ones by always taking a corresponding reacting transition. In particular, condition (1) ensures that processes in the

same state as the process taking the acting transition on some event e have a way to react to e. Condition (2) ensures that, if a process can reach a state with a reacting transition on an initiable event in that phase, all other processes can also reach a state where they can react to that event. Condition (3) ensures that once a process takes an acting transition of event e and moves to a state s' where a reacting transition of event f can be taken, if f is initiable in the phase of s', all processes move in a way that ensures they can take a reacting transition of f as well.

Permissible Safety Specification. We target safety specifications which forbid the reachability of any global state where some number m (or more) of processes are in some set of local states; simple instantiations of such specifications include mutual-exclusion and process-local safety properties. Let f be a Boolean formula over locations and **int** variables of a Mercury process. Let f_i be f indexed by the PID i. For instance, $f = s.loc \neq c \ \wedge \ s.\sigma(v) < 1$ has the indexed formula $f_i = q[i].loc \neq c \ \wedge \ q[i].\sigma(v) < 1$. Let $i_1, \ldots, i_m$ represent distinct valuations over I_n. Then, we define $\phi_{m,f}(n)$ as:

$$\phi_{m,f}(n) = \forall i_1, \ldots, i_m. \ \neg \left(f_{i_1} \wedge \ldots \wedge f_{i_m}\right).$$

Intuitively, the formula f encodes a set $st(f) = \{s \in S \mid f = true\}$ of process-local states (where f holds) and the property $\phi_{m,f}(n)$ forbids the reachability of a global state where m or more processes are in the set of local states $st(f)$. We call formulas of the form $\phi_{m,f}(n)$ permissible safety specifications and note that all specifications in this paper can be expressed using this form. For example, the Distributed Store specification asserting that no more than 1 process is in location Leader is expressed as $\phi_{2,s.loc=\texttt{Leader}}(n)$, i.e., $\forall i_1, i_2. \ \neg \left(q[i_1].loc = \texttt{Leader} \wedge q[i_2].loc = \texttt{Leader}\right)$.

Examples of specifications that are *not* permissible include: "there exists at least one process in location Leader at all times", and, "no more than half of the processes can be in location Leader". The former forbids m or less processes to be in a given set of local states (as opposed to m or more) while the latter forbids the reachability of a possibly unbounded number of processes to a given set of states (as opposed to a fixed number m of processes). For the remainder of this section, we will focus on permissible safety specifications of the form $\phi_{m,f}(n)$, but we note that our results extend to conjunctions and disjunctions of permissible safety specifications (ref. extended version [Jaber et al. 2020b]).

On a high-level, if a Mercury process is phase-compatible, then the behavior of the corresponding Mercury system is independent of its number of processes. Hence, the reachability, or the lack thereof, of an error state corresponding to a violation of a permissible safety specification is consistent across different "sizes" of the system. In other words, if an error state is reachable in a system with a given number of phase-compatible processes, adding additional processes will not render such an error state unreachable. Decidability follows from a similar argument in the opposite direction: if an error state is reachable in a system with some number of phase-compatible processes, then we can compute a number of processes sufficient for reaching the error state.

The following theorem identifies the decidable fragment for MPVP.

Theorem 4.3. *MPVP is decidable for Mercury system process P and permissible safety specification $\phi(n)$ if:*

(1) P is phase-compatible.
(2) The state space of P is fixed and finite[9].

[9]We note that this condition restricts the way participant sets of agreement primitives are built to the constant set **All** or the result of a previous **Partition** instance part (part.**winS** or part.**loseS**), hence ensuring the precondition of agreement is naturally met. In general, this condition can be relaxed to include some systems with an unbounded state space where such sets are built through communication.

(3) There exists at most one rendezvous-receive transition per action per phase[10].

Proof Intuition. The proof leverages a new fragment of an existing abstract model, the GSP model [Jaber et al. 2020a], for which MPVP is decidable. The decidability result for the GSP model utilizes the framework of *well-structured transition systems* (WSTSs). This entails defining a well-quasi-ordering (WQO) over the global state space of a GSP system as well as a set of sufficient "well-behavedness" conditions over the local GSP process definition to ensure that global transitions are "compatible" with the well-quasi-ordering. To admit a larger decidable fragment, we designed a novel WQO and relaxed these well-behavedness conditions. Further, we model process crash-stop failures. We defer the intricacies of this extension (which we refer to as the Mercury Core), as well as the formal definitions of WSTSs, WQOs, and compatibility to the extended version [Jaber et al. 2020b]. We note that, without this extension, the phase-compatibility conditions will not be initiability-aware (e.g., phase-compatibility condition (2) above will need to hold regardless of e being initiable or not).

We show that for any Mercury process P_{Merc} that satisfies the conditions of Theorem 4.3, one can construct a corresponding process P_{Core} in Mercury Core such that (i) there exists a simulation equivalence between their respective global state-transition systems and (ii) P_{Core} is in the decidable fragment of Mercury Core. We refer the interested reader to the extended version [Jaber et al. 2020b] for the full proof.

Recall that, in Sec. 1.2, we discuss reasons why neither the decidable fragment of GSP model nor Mercury Core is directly suitable for designing agreement-based decidable systems.

4.2 Cutoffs for Efficient Parameterized Verification

We define additional conditions on Mercury programs to obtain small cutoffs and enable efficient parameterized verification. These *cutoff-amenability conditions* ensure that any global error state, where m processes are in local states $st(f)$ violating a permissible safety specification $\phi_{m,f}(n)$, can be *reached* in a system with exactly m processes iff it can be reached in a system of any size larger than m. Thus, programs satisfying these conditions enjoy a small model property: $\phi_{m,f}(n)$ is satisfied in $\mathcal{M}(n)$ for all $n \in \mathbb{N}$ if $\phi_{m,f}(m)$ is satisfied in a system $\mathcal{M}(m)$ with a fixed number of processes m. This requires the conditions to ensure that the reachability of a global state violating $\phi_{m,f}(m)$ in $\mathcal{M}(m)$ *does not depend on the existence of more than m processes.*

Cutoff-Amenability Conditions. We first define a notion of *independence* of transitions and paths of a process. Informally, independent transitions do not require the *existence* of other processes in certain states. For instance, in **Partition** agreement, the winning transition $s \xrightarrow{\texttt{win}:PC_{\texttt{part}}(\texttt{pcpt},k)} s'$ is independent since a winning process does not require the existence of a losing one to take that transition, but the losing transition $s \xrightarrow{\texttt{lose}:PC_{\texttt{part}}(\texttt{pcpt},k)} s'$ is not independent since the losing process requires the existence of a winning process to take that transition. Note that acting transitions of globally-synchronizing events as well as internal transitions are independent while reacting transitions are not independent. A path is independent if it consists of independent transitions.

Definition 4.4 (Cutoff-Amenability Conditions). Let P be a phase-compatible process, $\phi_{m,f}(n)$ a permissible specification, and $\mathcal{F}$ the set of independent simple paths from s_0 to a state $s \in st(f)$. We require either of the following to hold.

(1) All paths from s_0 to $st(f)$ are independent, or,
(2) For every transition $s_s \rightarrow s_d$ such that $s_s \neq s_d$ and s_s is a state in some path $p \in \mathcal{F}$, either

[10]Under full symmetry, this condition ensures that abstracting the receiver PID (in any rendezvous-send transition $s \xrightarrow{\textbf{sendrz}(\text{act},PID)} s'$) does not introduce spurious behaviors.

(a) the state s_d is in p and the transition $s_s \rightarrow s_d$ is independent, or,
(b) the state s_d is not in p and all paths out of s_d lead back to s_s via independent transitions.

The conditions ensure that the processes required to *enable* a path to an error state are available in $\mathcal{M}(m)$. Condition (1) ensures that, if m processes were to reach the error states, they can do so without requiring additional processes, since all paths to the error states are independent. Condition (2) allows for some processes to "diverge" from the independent paths as long as they *return* independently. We note that the QuickSilver tool implements a more advanced version of this lemma that allows for more systems to have cutoffs.

We refer to the pair $\langle P, \phi_{m,f}(n)\rangle$ as *amenable* if P is a phase-compatible process that satisfies the cutoff-amenability conditions w.r.t. permissible safety specification $\phi_{m,f}(n)$.

On a high-level, an amenable pair $\langle P, \phi_{m,f}(n)\rangle$ identifies systems where the minimum number of processes to trigger an error (i.e., m process existing simultaneously in $st(f)$) is, in fact, exactly m. This is achieved by ensuring that any path a process may take to an error state is independent and hence if a process may reach an error state, it can do so without the help of other processes.

Lemma 4.5. *For an amenable pair $\langle P, \phi_{m,f}(n)\rangle$, $c = m$ is a cutoff for MPVP.*

Proof Intuition. We utilize the cutoff results of Mercury Core. Using the construction in the proof of Theorem 4.3 to obtain a process P_{Core} in the Mercury Core from a process P_{Merc} in Mercury, we show that if cutoff-amenability holds for P_{Merc}, then P_{Core} will be cutoff-amenable and the resulting cutoff for $\langle P_{\text{Core}}, \phi_{m,f}(n)\rangle$ is also a cutoff for $\langle P_{\text{Merc}}, \phi_{m,f}(n)\rangle$. We refer the reader to the extended version [Jaber et al. 2020b] for the full proof.

Automation and Feedback. While the phase-compatibility and cutoff-amenability conditions are somewhat intricate, we emphasize that our QuickSilver tool automatically checks these conditions and additionally gives the system designer feedback on how to make a process phase-compatible and amenable. This allows the designer to proceed without being caught up in the details of the exact conditions. The feedback varies depending on the failed condition and mainly aims to capture the root cause of the failure and to provide heuristically-ranked suggestions to fix it.

A failure of a phase-compatibility condition can be succinctly captured by a phase, a set of local states in that phase, and set of acting/reacting transitions from these states over one or two events. This localization is valuable for the user to pin-point what changes are needed to render the system phase-compatible. QuickSilver suggests edits that would eliminate the current violation and the user gets to pick which edit to implement.

A failure of a cutoff-amenability condition for correctness properties $\phi_{m,f}(n)$ can be succinctly captured by a non-independent path from the initial state s_0 to a state $s \in states(f)$. This path indicates a scenario where an error state could be unreachable in a system with m processes but can be reachable in a bigger system; hence m is not a valid cutoff. In these cases, QuickSilver presents the non-independent paths but does not suggest edits, and the user is responsible for eliminating the non-independent transitions from these paths.

Example. Consider a system where a set of processes wish to select up to two processes that can then perform an action one after the other. The designer starts with the process definition shown in Fig. 6. All processes start in location `Start`, and coordinate to pick up to two processes to move to the `Selected` location while the rest move to `Idle`. From `Selected`, the chosen processes send the `getReady` broadcast and move to `Prepare`. In `Prepare`, they attempt to move to the `Target` location one after the other using a `sequencer` message. The correctness property for this system is $\phi_{2,loc=\texttt{Target}}(n)$ indicating that at most one process can be at the `Target` location at any time.

```
process SelectiveSerializer
actions
  br getReady : unit
  br sequencer : unit

 initial location Start
   on Partition<select>(All,2)
     win: goto Selected
     lose: goto Idle

location Idle
   passive getReady, sequencer

 location Selected
   on _ do
     sendbr(getReady)
     goto Prepare

location Prepare
   on recv(sequencer) do
     goto Target

location Target
  // perform action
```

Fig. 6. An Initial Mercury Process Definition for Distributed Coordination for Serializing Access.

When QuickSilver in run on this process definition, it reports that the system is not phase-compatible with the following feedback suggesting adding a receive handler of event `getReady` from the `Selected` location:

```
(Selected,{}) needs a corresponding reacting transition on getReady
Suggestions to solve this:
- add transition (Selected,{}) ──R(getReady)──> (Prepare,{})
- add transition (Selected,{}) ──R(getReady)──> (Anywhere!,{})
```

The designer accepts the first heuristically-ranked suggestion and adds the following handler to the `Selected` location:

```
on recv(getReady) do goto Prepare
```

With this edit, the system is now phase-compatible. However, the system is not cutoff-amenable. QuickSilver returns the following feedback:

```
Cutoff computation failed: on path
(Start,{})──A(select)──>(Selected,{})──A(getReady)──>(Prepare,{})──R(sequencer)──>(Target,{})
the following transition(s) are not independent:
(Prepare,{})──R(sequencer)──> (Target,{})
```

Based on this feedback, the designer realizes that processes in `Prepare` need to send the `sequencer` broadcast to move to the `Target` location, which is an independent transition. Learning from the previous phase-compatibility violation, the user additionally adds the corresponding reacting transition. Thus, the designer replaces the receive handler from location `Prepare` with the following two handlers:

```
on _ do sendbr(sequencer) goto Target
```

```
on recv(sequencer) do goto Prepare
```

The system is now phase-compatible and cutoff-amenable, and QuickSilver reports a cutoff value of two.

Modular Verification. Recall that Lemma 3.1 shows that any verified agreement protocol (i.e., one proven to satisfy agreement, validity, and termination) also meets the pre- and post-condition pair of our agreement primitives. So, by verifying an Mercury program with agreement primitives w.r.t.

to a safety specification, we can conclude that the program, when instantiated with any agreement protocol that satisfies agreement, validity, and termination, also satisfies the safety specification.

5 IMPLEMENTATION AND EVALUATION

We describe the implementation of QuickSilver[11] and evaluate the performance of its automated parameterized verification procedure on various benchmarks encoded in Mercury.

5.1 Implementation

QuickSilver performs automated, parameterized verification of Mercury programs in three steps:

1. *Parsing.* QuickSilver compiles Mercury processes into the core fragment by rewriting all non-core handlers (e.g., handlers with if-statements or multiple send statements) into core handlers and expanding syntactic sugar.
2. *Analysis.* From the core fragment, QuickSilver creates a labeled graph representing the process-level semantics including transitions that model crash-stop failures. QuickSilver checks the phase-compatibility and cutoff-amenability conditions against this graph, and if the conditions are met, computes a cutoff to verify the system.
3. *Verification.* QuickSilver's verification engine is built on top of Kinara [Alur et al. 2015], a verification tool for distributed systems with a *fixed* number of processes. QuickSilver extends Kinara to support the **`Partition`** and **`Consensus`** primitives as well as their global behaviors. QuickSilver translates the core fragment of Mercury into the input representation accepted by the extended version of Kinara using the cutoff number of processes, as computed during the analysis step. Permissible safety specifications $\phi_{m,f}(n)$ are encoded in QuickSilver as `atmost(`$m-1$`,{loc:`$\langle bExp \rangle$`})` where the Boolean expression f is such that $\forall s \in states(f) : s.loc = \texttt{loc} \wedge s.\sigma(bExp) = \texttt{True}$. For example, the property $\phi_{3,loc=\texttt{Replica} \wedge stored=1}(n)$ is encoded as `atmost(2 ,{Replica: stored == 1}).` The environment process is automatically generated to nondeterministically send/receive all environmental communication actions that the Mercury process expects. The specifications as well as the environment process are translated to Kinara's representation similarly. QuickSilver reports successful parameterized verification iff Kinara reports successful verification for the system consisting of the cutoff number of processes.

User Feedback. QuickSilver helps the user obtain a phase-compatible and cutoff-amenable process by providing heuristically ranked suggestions to handle any violation of the phase-compatibility and cutoff-amenability conditions during the analysis step. For instance, the phase-compatibility conditions do not hold if a local state has an acting transition but not its corresponding reacting transition. In this case, QuickSilver returns the violated condition and suggests adding a handler that corresponds to the reacting transition from that state.

5.2 Evaluation

The research questions we tackle in this evaluation are:

RQ1 Can interesting agreement-based systems be modeled concisely in Mercury?
RQ2 Can interesting agreement-based systems be modeled in the decidable fragment of Mercury with relative ease?
RQ3 Can QuickSilver perform automated parameterized verification of agreement-based systems in Mercury in a reasonable amount of time?
RQ4 Do QuickSilver's cutoffs enable efficient verification?

In what follows, we first present our Mercury benchmarks and address RQ1 and RQ2. We then analyze the performance of QuickSilver on these benchmarks and address RQ3 and RQ4. All

[11] A virtual machine containing QuickSilver is publicly available [QuickSilver 2021].

experiments were performed on an Intel Xeon machine with E5-2690 CPU and 32GB of RAM. We report the mean run time for 10 runs as well as the 95% confidence interval for each benchmark.

Mercury Benchmarks. Our benchmarks are briefly described below.[12]

1. *Distributed Store* is the illustrative example from Sec. 2.1.
2. *Consortium* is a distributed system where a set of actors wants to reach a decision based on information the actors gather individually. A subset of the actors is elected and trusted with making a decision that is then announced to the rest of the actors. This resembles scenarios where a trade-off between trust and performance is needed (e.g., a consortium blockchain [Amsden et al. 2020; Hyperledger 2021]). The safety property for this system is (1) at most two actors are elected to decide on a value for all processes and (2) that all actors agree on the decided value.
3. *Two-Object Tracker* is a system for collaborative surveillance based on leader election and is inspired by an example in [Chang and Tsai 2016]. Upon detecting an object, a leader is elected to be responsible for monitoring it along with its followers. The system can additionally fork another set of processes to monitor a second object simultaneously. The safety property for Two-Object Tracker is that there can be at most two leaders at a time, and when a second object is spotted, each of the leaders is tracking a distinct object.
4. *Distributed Robot Flocking* is a distributed system where processes follow a common leader as a flock and is inspired by an example in [Canepa and Potop-Butucaru 2007]. Processes can disperse into various locations where they can elect a leader. The leader then issues directions to the rest of the flock. This is especially useful in self-stabilizing systems. The safety property for this system is that the system stabilizes by ensuring there can be at most one leader at a time making direction decisions.
5. *Distributed Lock Service* is a distributed lock service similar to Chubby [Burrows 2006] for coarse-grained synchronization with an elected leader handling clients requests. Clients can interface with this lock service as a file system where they send reads and writes to an elected leader, and have their requests replicated safely on different servers. The leader periodically times out, sends a step down signal to the rest of the servers, and a new round of election is used to pick a new leader. The safety property for this system is that at most one server is elected as a single point-of-contact for the clients.
6. *Distributed Sensor Network* is a sensor network application that elects a subset of processes, who have sensed an environmental signal, to report to a centralized monitor. In this application, the set of sensors that have detected the environmental signal (and hence need to coordinate) is dynamically built before invoking the agreement protocol. The safety property here is that the environmental signal is reported by no more than two sensors.
7. *Sensor Network with Reset* is a variant of the Distributed Sensor Network benchmark that uses a "reset" signal to resume monitoring for the environmental signal, thereby requiring an unbounded number of rounds of agreement. The safety property is as before.
8. *Small Aircraft Transportation System (SATS) Landing Protocol* is the landing protocol of SATS proposed by NASA [2021]. The goal of SATS is to increase access to small airports without control towers by allowing aircraft to coordinate with each other to operate safely upon entering the airport airspace. For the landing protocol, the aircraft coordinate to choose successive subsets of aircraft to progress to the next phase of landing, until just one aircraft is chosen to land at a time. The desired safety properties for the SATS landing protocol, provided by NASA, are as follows: (1) there are a total of at most four aircraft across the airport vicinity; (2) there are a total of at most two aircraft across each left (right) holding zone of the airport; and (3) there is at most one aircraft that can attempt a final approach (i.e., attempt landing) at a time.

[12] Mercury code for benchmarks available at: https://tinyurl.com/m3zx7jxs

9. *SATS Landing Protocol II* is a version of the SATS landing protocol where aircrafts communicate explicitly to build a participant set when nearing the final approach, and return to a specific holding zone if they miss landing. The safety property is as before.
10. *Mobile Robotics Motion Planning* is a distributed system based on an existing benchmark [Desai et al. 2017] where a set of robots share a workspace with obstacles, and need to coordinate their movements. The robots coordinate to create a motion plan by successively choosing each robot to create a plan while taking into account the previous robots' plans. The targeted property for this system is collision avoidance; this is achieved by allowing the robots to create their motion plans consecutively one-at-a-time.
11. *Mobile Robotics with Reset* is a variant of the Mobile Robotics Motion Planning benchmark that allows all the robots to return to their initial state upon receiving a signal to do so. The safety property is as before.
12. *Distributed Register* is a data store á la Atomix's AtomicValue [Atomix 2021] which gives a consistent view of a stored value under concurrent updates. Updates that do not change the stored data in the register are ignored. The safety property for the Distributed Register system is that no two replicas that are about to serve user requests may have different values of the register; hence ensuring the clients have a consistent view of the data.

RQ1. Our benchmarks are models of distributed agreement-based systems commonly found in the literature and have all been encoded in Mercury with relative ease by the authors of this work. Further, as can be seen in Column 2 of Table 1, the corresponding Mercury process definitions are fairly compact, i.e., within 100 lines of code (LoC). Thus, our answer to RQ1 is Yes.

RQ2. We found two factors valuable in addressing RQ2.
Value of User Feedback. It is not always easy for a system designer to ensure that their initial model of a Mercury process is phase-compatible. For example, when modeling the Distributed Lock Service benchmark, we made assumptions about the behavior of the system, causing us to omit reacting transitions on some events and, consequently, our initial model was not phase-compatible. However, the feedback provided by QuickSilver helped identify the missing transitions that needed to be added.

Value of Mercury Core. Prior decidability results did not encompass all of the benchmarks we evaluate; in particular, those marked with * in Table 1 fall outside prior known fragments. Mercury Core's extension of decidability results, on the other hand, enables decidable parameterized verification for all of our benchmarks.

Thus, with the help of QuickSilver's user feedback and the Mercury Core decidable fragment, our answer to RQ2 is Yes.

RQ3. In Table 1, for each benchmark we provide the number of phases, the cutoff used for verification, and the mean run time of QuickSilver with its 95% confidence intervals. Notice that the cutoffs computed by QuickSilver for all benchmarks are small (under 6 processes). Overall, QuickSilver performs efficient parameterized verification for all benchmarks, taking less than 2 seconds to verify most benchmarks, and about 12 minutes for the largest benchmark, SATS Landing Protocol II. Thus, our answer to RQ3 is also Yes.

RQ4. To examine the contribution of cutoffs in enabling efficient verification, we performed experiments studying the effect of varying the number of processes on the run time of QuickSilver. As expected, increasing the number of processes causes the run time to grow exponentially. For instance, the time to verify the Consortium benchmark jumps from 9 seconds to about 8 minutes when verifying a system with 5 processes instead of the 3-process cutoff. Fortunately, QuickSilver is able to detect small cutoffs to sidestep the exponential growth caused by increasing the number of processes, enabling practical parameterized verification. Thus, our answer to RQ4 is Yes.

Table 1. QuickSilver Performance.

Benchmark	LoC	Phases	Cutoff	Time(s)
Distributed Store	64	2	3	45.079 ± 0.621
Consortium*	58	9	3	6.953 ± 0.022
Two-Object Tracker*	69	9	3	0.641 ± 0.006
Distributed Robot Flocking	78	7	2	0.105 ± 0.002
Distributed Lock Service	38	2	2	0.059 ± 0.002
Distributed Sensor Network	55	3	3	1.041 ± 0.003
Sensor Network with Reset	63	3	3	1.662 ± 0.012
SATS Landing Protocol*	90	3	5	638.393 ± 0.872
SATS Landing Protocol II*	99	5	5	736.417 ± 3.659
Mobile Robotics Motion Planning	71	5	2	0.114 ± 0.004
Mobile Robotics with Reset*	83	4	2	0.166 ± 0.003
Distributed Register	32	1	2	0.329 ± 0.006

Remark. QuickSilver additionally reports the number of phases, which correspond to global guards in the Mercury Core, that QuickSilver automatically generates. This shows the value of automation as designing such guards manually is tedious and error-prone.

6 CONCLUDING REMARKS

We presented a framework, QuickSilver, for modeling and efficient, automated parameterized verification of agreement-based systems. The framework supports a modular approach to the design and verification of distributed systems in which systems are (i) modeled using sound abstractions of complex distributed components and (ii) verified using model checking-based techniques assuming that the complex components are verified separately, presumably using deductive techniques.

In ongoing work, we focus on extending QuickSilver to handle non-blocking communication and network failures using "channels" that can buffer or drop messages, to support infinite variable domains using abstract interpretation, and to help system designers *synthesize* amenable processes.

Eventually, we hope to see this framework generalized by us or our readers to other verified distributed components and richer properties such as liveness. We also hope to see more conversations and verification frameworks, in particular layered ones, that cut across the deductive verification and model checking communities.

ACKNOWLEDGMENTS

We thank Ilya Sergey, Rupak Majumdar, Isil Dillig, Thomas Wahl, and Marijana Lazic for their invaluable feedback on various drafts of this paper, and Derek Dreyer for helping us interpret reviewer comments about an earlier draft. We are grateful to Shaz Qadeer and Ken McMillan for their thought-provoking questions at earlier stages of this work that helped shape this paper. We also thank Abhishek Udupa for patiently answering all our questions about the Kinara tool. Finally, the authors are grateful to the anonymous reviewers from POPL 2019, CAV 2019, POPL 2020, CAV 2020, POPL 2021, PLDI 2021, and OOPSLA 2021 who chose to take the time to provide constructive feedback on our submissions. This research was partially supported by the National Science Foundation under Grant Nos. 1846327, 1908504, and by grants from the Purdue Research Foundation and Amazon Science. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

Rajeev Alur, Milo Martin, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. 2014. Synthesizing Finite-State Protocols from Scenarios and Requirements. In *Hardware and Software: Verification and Testing*, Eran Yahav (Ed.). Springer International Publishing, Cham, 75–91.

Rajeev Alur, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. 2015. Automatic Completion of Distributed Protocols with Symmetry. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 395–412.

Rajeev Alur and Stavros Tripakis. 2017. Automatic Synthesis of Distributed Protocols. *SIGACT News* 48, 1 (March 2017), 55–90. https://doi.org/10.1145/3061640.3061652

Benjamin Aminof, Tomer Kotek, Sasha Rubin, Francesco Spegni, and Helmut Veith. 2018. Parameterized model checking of rendezvous systems. *Distributed Computing* 31, 3 (2018), 187–222. https://doi.org/10.1007/s00446-017-0302-6

Zachary Amsden, Ramnik Arora, Shehar Bano, Mathieu Baudet, Sam Blackshear, Abhay Bothra, George Cabrera andChristian Catalini, Konstantinos Chalkias, Evan Cheng, Avery Ching, Andrey Chursin, George Danezis andGerardo Di Giacomo, David L. Dill, Hui Ding, Nick Doudchenko, Victor Gao, Zhenhuan Gao, François Garillot, Michael Gorven, Philip Hayes, J. Mark Hou, Yuxuan Hu, Kevin Hurley, Kevin Lewi, Chunqi Li, Zekun Li, Dahlia Malkhi andSonia Margulis, Ben Maurer, Payman Mohassel, Ladi de Naurois, Valeria Nikolaenko, Todd Nowacki, Oleksandr Orlov andDmitri Perelman, Alistair Pott, Brett Proctor, Shaz Qadeer, Rain, Dario Russi, Bryan Schwab, Stephane Sezer, Alberto Sonnino, Herman Venter, Lei Wei, Nils Wernerfelt, Brandon Williams, Qinfan Wu, Xifan Yan, Tim Zakian, and Runtian Zhou. 2020. *The Libra Blockchain.* Technical Report. https://developers.libra.org/docs/assets/papers/the-libra-blockchain/2020-05-26.pdf

Kristoffer Just Arndal Andersen and Ilya Sergey. 2019. Distributed Protocol Combinators. In *Practical Aspects of Declarative Languages*, José Júlio Alferes and Moa Johansson (Eds.). Springer International Publishing, Cham, 169–186.

Krzysztof R. Apt and Dexter C. Kozen. 1986. Limits for automatic verification of finite-state concurrent systems. *Inform. Process. Lett.* 22, 6 (1986), 307–309. https://doi.org/10.1016/0020-0190(86)90071-2

A. Arghavani, E. Ahmadi, and A. T. Haghighat. 2011. Improved bully election algorithm in distributed systems. In *ICIMU 2011 : Proceedings of the 5th international Conference on Information Technology Multimedia.* 1–6. https://doi.org/10.1109/ICIMU.2011.6122724

Atomix. 2021. Atomix. https://atomix.io/docs/latest/user-manual/primitives/AtomicValue/

Simon Außerlechner, Swen Jacobs, and Ayrat Khalimov. 2016. Tight Cutoffs for Guarded Protocols with Fairness. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9583)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, 476–494. https://doi.org/10.1007/978-3-662-49122-5_23

Roderick Bloem, Nicolas Braud-Santoni, and Swen Jacobs. 2016. Synthesis of Self-Stabilising and Byzantine-Resilient Distributed Systems. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 157–176.

Mike Burrows. 2006. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington) *(OSDI '06)*. USENIX Association, USA, 335–350.

Davide Canepa and Maria Gradinariu Potop-Butucaru. 2007. Stabilizing Flocking Via Leader Election in Robot Networks. In *Stabilization, Safety, and Security of Distributed Systems*, Toshimitsu Masuzawa and Sébastien Tixeuil (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–66.

Saksham Chand, Yanhong A. Liu, and Scott D. Stoller. 2016. Formal Verification of Multi-Paxos for Distributed Consensus. In *FM 2016: Formal Methods*, John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou (Eds.). Springer International Publishing, Cham, 119–136.

Tushar D Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos Made Live: an Engineering Perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. ACM, 398–407.

Che-Cheng Chang and Jichiang Tsai. 2016. Distributed collaborative surveillance system based on leader election protocols. *IET Wireless Sensor Systems* 6, 6 (2016), 198–205. https://doi.org/10.1049/iet-wss.2015.0030

Bernadette Charron-Bost and André Schiper. 2009. The Heard-of Model: Computing in Distributed Systems with Benign Faults. *Distributed Computing* 22, 1 (2009), 49–71. https://doi.org/10.1007/s00446-009-0084-6

Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. 2012. TLA+ Proofs. In *International Symposium on Formal Methods*. Springer, 147–154.

Andrei Damian, Cezara Dragoi, Alexandru Militaru, and Josef Widder. 2019. Communication-closed Asynchronous Protocols. In *International Conference on Computer Aided Verification*.

Werner Damm and Bernd Finkbeiner. 2014. Automatic Compositional Synthesis of Distributed Systems. In *International Symposium on Formal Methods*. Springer, 179–193.

Giorgio Delzanno, Jean-François Raskin, and Laurent Van Begin. 2002. Towards the Automated Verification of Multithreaded Java Programs. In *TACAS (Lecture Notes in Computer Science, Vol. 2280)*. Springer, 173–187.

Ankush Desai, Indranil Saha, Jianqiao Yang, Shaz Qadeer, and Sanjit A. Seshia. 2017. DRONA: A Framework for Safe Distributed Mobile Robotics. In *Proceedings of the 8th International Conference on Cyber-Physical Systems* (Pittsburgh, Pennsylvania) *(ICCPS '17)*. ACM, 239–248.

Ryan Doenges, James R Wilcox, Doug Woos, Zachary Tatlock, and Karl Palmskog. 2017. Verification of Implementations of Distributed Systems Under Churn. (2017).

Cezara Drăgoi, Thomas A Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. 2014. A Logic-based Framework for Verifying Consensus Algorithms. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 161–181.

Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. 2016. PSync: A Partially Synchronous Language for Fault-Tolerant Distributed Algorithms. *SIGPLAN Not.* 51, 1 (Jan. 2016), 400–415. https://doi.org/10.1145/2914770.2837650

E. Allen Emerson and Vineet Kahlon. 2003a. Exact and Efficient Verification of Parameterized Cache CoherenceProtocols. In *CHARME (Lecture Notes in Computer Science, Vol. 2860)*. Springer, 247–262.

E. Allen Emerson and Vineet Kahlon. 2003b. Model Checking Guarded Protocols. In *18th IEEE Symposium on Logic in Computer Science (LICS 2003), 22-25 June 2003, Ottawa, Canada, Proceedings*. IEEE Computer Society, 361–370.

E. Allen Emerson and A Prasad Sistla. 1996. Symmetry and Model Checking. *Formal methods in system design* 9, 1-2 (1996), 105–131.

E. Allen Emerson and Thomas Wahl. 2003. On Combining Symmetry Reduction and Symbolic Representation for Efficient Model Checking. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer, 216–230.

Javier Esparza, Alain Finkel, and Richard Mayr. 1999. On the Verification of Broadcast Protocols. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*. IEEE Computer Society, 352–359. https://doi.org/10.1109/LICS.1999.782630

Yotam M. Y. Feldman, James R. Wilcox, Sharon Shoham, and Mooly Sagiv. 2019. Inferring Inductive Invariants from Phase Structures. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 405–425.

Hector Garcia-Molina. 1982. Elections in a distributed computing system. *IEEE Computer Architecture Letters* 31, 01 (1982), 48–59. https://doi.org/10.1109/TC.1982.1675885

Álvaro García-Pérez, Alexey Gotsman, Yuri Meshman, and Ilya Sergey. 2018. Paxos Consensus, Deconstructed and Abstracted. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 912–939.

Steven M. German and A. Prasad Sistla. 1992. Reasoning about Systems with Many Processes. *J. ACM* 39, 3 (July 1992), 675–735. https://doi.org/10.1145/146637.146681

Jeremiah Griffin, Mohsen Lesani, Narges Shadab, and Xizhe Yin. 2020. TLC: Temporal Logic of Distributed Components. *Proc. ACM Program. Lang.* 4, ICFP, Article 123 (Aug. 2020), 30 pages. https://doi.org/10.1145/3409005

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/2815400.2815428

Hyperledger. 2021. The Hyperledger Project. https://www.hyperledger.org/

C Norris Ip and David L Dill. 1996. Better Verification Through Symmetry. *Formal methods in system design* 9, 1-2 (1996), 41–75.

Nouraldin Jaber, Swen Jacobs, Christopher Wagner, Milind Kulkarni, and Roopsha Samanta. 2020a. Parameterized Verification of Systems with Global Synchronization and Guards. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 299–323.

Nouraldin Jaber, Christopher Wagner, Swen Jacobs, Milind Kulkarni, and Roopsha Samanta. 2020b. Parameterized Reasoning for Distributed Systems with Consensus. *CoRR* abs/2004.04613 (2020). arXiv:2004.04613 https://arxiv.org/abs/2004.04613

Swen Jacobs and Mouhammad Sakr. 2018. Analyzing Guarded Protocols: Better Cutoffs, More Systems, More Expressivity. In *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10747)*, Isil Dillig and Jens Palsberg (Eds.). Springer, 247–268. https://doi.org/10.1007/978-3-319-73721-8_12

Bernhard Kragl, Constantin Enea, Thomas A. Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. 2020. Inductive Sequentialization of Asynchronous Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 227–242. https://doi.org/10.1145/3385412.3385980

Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 336–365.

Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. https://doi.org/10.1145/279227.279229

Leslie Lamport. 2002. *Specifying Systems: the TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Longman Publishing Co., Inc.

Leslie Lamport. 2006. Fast paxos. *Distributed Computing* 19, 2 (2006), 79–103. https://doi.org/10.1007/s00446-006-0005-x

Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (Dec. 1975), 717–721. https://doi.org/10.1145/361227.361234

Yanhong A. Liu, Scott D. Stoller, Bo Lin, and Michael Gorbovitski. 2012. From Clarity to Efficiency for Distributed Algorithms. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 395–410. https://doi.org/10.1145/2384616.2384645

Nancy A. Lynch. 1996. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 369–384. https://doi.org/10.5555/1855741.1855767

Ognjen Marić, Christoph Sprenger, and David Basin. 2017. Cutoff Bounds for Consensus Algorithms. In *International Conference on Computer Aided Verification*. Springer, 217–237.

NASA. 2021. NASA - Small Aircraft Transportation System. https://www.nasa.gov/centers/langley/news/factsheets/SATS.html

Diego Ongaro and John K Ousterhout. 2014. In Search of an Understandable Consensus Algorithm.. In *USENIX Annual Technical Conference*. 305–319.

Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. 2017a. Reducing Liveness to Safety in First-Order Logic. *Proc. ACM Program. Lang.* 2, POPL, Article 26 (Dec. 2017), 33 pages. https://doi.org/10.1145/3158114

Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. 2017b. Paxos Made EPR: Decidable Reasoning about Distributed Protocols. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 108 (Oct. 2017), 31 pages. https://doi.org/10.1145/3140568

Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: Safety Verification by Interactive Generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 614–630. https://doi.org/10.1145/2908080.2908118

Ruzica Piskac, Leonardo de Moura, and Nikolaj Bjørner. 2010. Deciding Effectively Propositional Logic Using DPLL and Substitution Sets. *Journal of Automated Reasoning* 44, 4 (2010), 401–424.

QuickSilver. 2021. QuickSilver Implementation. https://doi.org/10.5281/zenodo.5501650

Vincent Rahli. 2012. Interfacing with Proof Assistants for Domain Specific Programming Using EventML. (2012).

RedisRaft. 2021. RedisRaft. https://github.com/RedisLabs/redisraft/

Sylvain Schmitz and Philippe Schnoebelen. 2013. The Power of Well-Structured Systems. In *CONCUR 2013 (Lecture Notes in Computer Science, Vol. 8052)*, Pedro R. D'Argenio and Hernán C. Melgratti (Eds.). Springer, 5–24. https://doi.org/10.1007/978-3-642-40184-8_2

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2017. Programming and Proving with Distributed Protocols. *Proc. ACM Program. Lang.* 2, POPL, Article 28 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158116

Ichiro Suzuki. 1988. Proving Properties of a Ring of Finite-State Machines. *Inf. Process. Lett.* 28, 4 (July 1988), 213–214. https://doi.org/10.1016/0020-0190(88)90211-6

Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. 2018. Modularity for Decidability of Deductive Verification with Applications to Distributed Systems. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 662–677. https://doi.org/10.1145/3192366.3192414

Klaus v. Gleissenthall, Rami Gökhan Kıcı, Alexander Bakst, Deian Stefan, and Ranjit Jhala. 2019. Pretend Synchrony: Synchronous Verification of Asynchronous Distributed Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 59 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290372

Thomas Wahl. 2007. Adaptive Symmetry Reduction. In *International Conference on Computer Aided Verification*. Springer, 393–405.

James R. Wilcox, Ilya Sergey, and Zachary Tatlock. 2017. Programming Language Abstractions for Modularly Verified Distributed Systems. In *2nd Summit on Advances in Programming Languages (SNAPL 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 71)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 19:1–19:12.

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 357–368. https://doi.org/10.1145/2737924.2737958

Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. 2016. Planning for Change in a Formal Verification of the Raft Consensus Protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs* (St. Petersburg, FL, USA) *(CPP 2016)*. Association for Computing Machinery, New York, NY, USA, 154–165. https://doi.org/10.1145/2854065.2854081

Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (Boston, Massachusetts) *(NSDI'09)*. USENIX Association, USA, 213–228.