Modularity for Decidability: Implementing and Semi-Automatically Verifying Distributed Systems

Anonymous Author(s)

Abstract

Proof automation can substantially increase productivity in the formal verification of complex systems. However, the unpredictablility of automated provers in handling quantified formulas presents a major hurdle to usability of these tools. Here, we propose to solve this problem not by improving the provers, but by using a modular proof methodology that allows us to produce *decidable* verification conditions. Decidability greatly improves the predictability of proof automation, resulting in a more practical verification approach. We apply this methodology to develop verified implementations of distributed protocols with competitive performance, showing its effectiveness.

1 Introduction

Distributed protocols play an essential role in important emerging technologies such as blockchains and autonomous vehicles. Testing such systems is not always effective, and the cost of failures can be large. This is well-recognized, e.g., in the ISO 26262 Functional Safety Standard [38]. Therefore, the systems community is increasingly receptive to formal methods [8, 9, 14, 17, 30, 40]. However, existing verification techniques are difficult to apply and require a large proof engineering effort. For example, the Verdi proof of the Raft protocol [33] includes 50000 lines of proof for 530 lines of implementation [41], which required several person-years.

Automated provers, such as the Z3 SMT solver [11] are capable of greatly reducing the verification effort, as seen, e.g., in the IronFleet project [17]. By annotating the program with invariants, the programmer reduces the proof to lemmas called *verification conditions* that can be discharged by the automated prover. In case these lemmas fail, the prover can sometimes produce counterexamples that explain the failure and allow the programmer to correct the annotations.

Unfortunately, the behavior of provers can be quite unpredictable, especially when applied to formulas with quantifiers, which are common when verifying distributed protocols. Since the problem presented to the prover is in general undecidable, it is no surprise that the prover sometimes diverges or produces inconclusive results on small instances, or suffers from the "butterfly effect", when a seemly irrelevant change in the input causes the prover to fail. As observed in the IronFleet project, SMT solvers can fail even on tiny examples. When this happens, the user has little information with which to decide how to proceed. This was identified in IronFleet as the main hurdle in the verification task.

Decidable Reasoning We address this problem here by showing how to verify the safety of realistic implementations of distributed systems using only *decidable* verification conditions. This is accomplished in part by an appropriate choice of language to express the program, as in previous work [13, 35], but also by using modular reasoning techniques to structure the correctness proof such that difference components can be proved using decidable fragments of first-order logic with various background theories. We will refer to this approach as *decidable decomposition*.

Because our prover is a decision procedure for the logical fragments we use, we can guarantee that in principle it will always terminate with a proof or a counter-model. In practice, decidability means that the behavior of the prover is much more predictable [34]. We observe in our evaluation that the human effort needed to achieve decidable decomposition is modest, and that we can develop and verify realistic implementations of distributed protocols without relying on the automated prover to solve undecidable problems.

A key aspect of our methodology is the separation between the abstract model and its implementation using concrete data structures. We prove system-level properties of the protocol using the abstract model and use these properties as a lemma in proving the implementation. This is important for obtaining stratification in the use of quantifiers. In addition, we use abstract datatypes to hide the theory needed to implement a data structure behind a set of abstract properties. The resulting decidable decomposition allows us to prove each module locally with decidable logics, even though a global proof would take us outside the decidable range.

At the end of the process, we compile the verified system to executable code (which uses a small set of trusted libraries, e.g., to implement a built-in array type). Our preliminary experience indicates that this can be used to generate reasonably efficient verified distributed systems.

Contributions The contributions of our paper are:

1. A new methodology, *decidable decomposition*, for verifying realistic distributed systems using existing modularity principles to decompose the verification into lemmas proved in (different) decidable logics.
2. Verified implementations of two popular distributed algorithms, Raft and Multi-Paxos, showing that proofs of practical systems naturally decompose into decidable sub-problems.

PL'17, January 01–03, 2017, New York, NY, USA
2017.

3. An evaluation of the verification effort and run-time performance of these systems, showing that the effort required by our methodology is an order of magnitude smaller than with previous methodologies such as Verdi or IronFleet, and the obtained performance is comparable.

2 Overview

In this section, we motivate and demonstrate our key ideas on a simple example.

2.1 Example: Toy Leader Election

Figure 1 shows pseudocode for a node that participates in a toy leader election protocol, in which a finite set of nodes decide on a leader. The set of nodes is a parameter of the system, which is determined at run time and remains fixed throughout each run of the protocol. Each node may propose itself as a candidate by sending a message to all other nodes. Nodes vote, by sending a response message, for the first candidate from which they receive a message. A leader is elected when it receives a majority of the votes. This protocol will get stuck in many cases without electing a leader. However, it suffices to demonstrate our verification methodology, since it is *safe*, i.e., at most one leader is elected. Furthermore, a variant of this protocol is an essential ingredient of both Raft and Multi-Paxos, which is used in many production systems, as well as in our evaluation.

The goal of the verification is to show that at most one leader is elected. Despite the simplicity of the property and the code, existing verification techniques cannot automatically prove that the code is correct when executed by an unbounded number of nodes. Even when the code is annotated with invariants, the corresponding verification conditions are expressed in undecidable logics. As a result, checking them with existing theorem provers such as Z3 [11] often results in divergence, and behaves unpredictably in general. Indeed, previous verification efforts in the systems community identified this as a major hurdle for verification [17]. The complexity arises due to the combination of arithmetic (e.g. number of nodes that voted for a candidate), set cardinalities, and quantifiers in the invariant that quantify over unbounded domains (e.g., expressing the fact that for every two nodes at most one is a leader), especially *non-stratified* quantifier alternations (see Section 3.2), which give rise to potentially infinitely many instantiations.

2.2 Approach

In this paper we present a methodology for developing and verifying implementations of distributed systems, whose performance is similar to other verified implementations (e.g. [41]) but reduces the human effort by an order of magnitude. Rather than starting with a low-level implementation (e.g., in C), we define a domain specific imperative language

```
1  already voted := false
2  voters := ∅
3
4  upon client_request() do {
5   if ¬already voted {
6    send request_vote_msg(self);
7    already voted := true
8    voters := {self}
9   }
10 }
11 upon recv(msg) do {
12  if msg.type = request_vote_msg
13          ∧¬already voted {
14   already voted := true;
15   send vote_msg(self, msg.src)
16  } else if msg.type = vote_msg {
17   voters := voters ∪ { msg.src}
18   if |voters| > N/2 { send leader_msg(self) }
19  }
20 }
```

Figure 1. Toy leader election pseudocode.

which permits effective (decidable) reasoning on the one hand, and straightforward compilation to efficient C++ code on the other hand.

Modular proofs Our method leverages modularity in the assume-guarantee style for decidable reasoning, by structuring the correctness proof such that different parts of the proof reason about different aspects of the system, and at different abstraction levels, enabling each of them to be carried in (possibly different) decidable logics. The decomposition has two benefits. First, it allows to reduce quantifier alternations, and eliminate bad quantification cycles. Second, it allows to check the verification conditions of each module using a different background theory (or none).

Patterns for obtaining modular proofs We identify two patterns that capture common programming languages practices, and allow for a natural decidable decomposition: 1) Data structures with first-order interfaces, and 2) Protocol designs as lemmas for proving the implementation. We used these patterns to construct modular proofs for our examples.

Data structures with first-order interfaces. Efficient implementations deploy concrete primitive types (e.g., integers), which call for using theories (e.g., arithmetic) to reason about them. Also, when it comes to distributed systems, the code of a process may be run by any number of nodes. It is therefore natural to reason about such systems in first-order logic, using quantifiers over unbounded domains (e.g., the set of nodes) and uninterpreted functions. However, the combination of quantifiers, uninterpreted functions and, e.g., arithmetic, often leads to undecidability. Encapsulating primitive types in a pure (uninterpreted) first-order logic interface decomposes the verification problem into the problem of reasoning in uninterpreted first-order logic, where the primitive types are opaque and only their interface is used, and reasoning about the low-level implementation, showing that it satisfies its interface, which can be done in decidable interpreted theories.

Protocol designs as lemmas for proving the implementation. Data structures with first-order interfaces simplify reasoning, but in general, do not suffice for verifying interesting distributed systems in decidable logics. The reason for this is that even the first-order reasoning problem is undecidable,

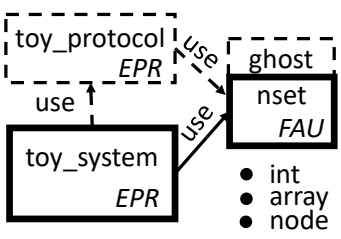

Figure 2. Modules and built-in types used for verifying the toy example. Ghost parts are depicted using dashed lines. Each module is annotated with the decidable fragment in which it was verified.

since proofs of complex systems require quantifier alternation, which leads to undecidability. Fortunately, distributed systems are constructed in an evolutionary process starting with a protocol design, and then gradually developing an efficient implementation, guided by the protocol design. In this paper, we exploit this idea in a novel way, by capturing the design and its correctness in a ghost object, which is verified separately and serves as a lemma for proving the implementation. This provides a natural way to further decompose the verification problem into sub-problems, where each sub-problem usually contains fewer quantifier alternations. In our limited experience, this usually results in each problem becoming decidable, i.e., contain only stratified (acyclic) quantifier alternations.

2.3 Modular Formulation

We illustrate our methodology on the simple leader election example, to verify that at most one leader is elected using *decidable* reasoning. Our formulation of the system consists of three *modules*: toy_protocol, toy_system, and nset. The interplay between the modules and the decidable fragments in which they are verified are depicted in Figure 2 (the fragments are defined in Section 3); their code is listed in Figures 3 to 5. The code is written in the **M**odular **D**ecidable **L**anguage (MDL) — a language for modular decidable verification and implementation of distributed systems.

The toy_protocol module presents (and verifies) an abstract model of the protocol, eliding some of the implementation details. The nset module defines (and verifies) a data type which encapsulates sets of nodes, and provides a first-order specification of the property that any two majorities intersect. The toy_system module specifies (and verifies) the system implementation, which uses both the data type (relying on its first order specification) and the abstract protocol (as ghost code) to obtain a verified executable implementation. We elaborate on these modules below.

Each module should be viewed as a proof unit, which consists of (1) declarations and definitions of types and state components, which may be interpreted using **interpret** declarations or remain uninterpreted, reflecting an abstraction of the actual theory, (2) assumptions about other modules, which are invariants verified in the other modules, specified by **use** declarations, (3) an invariant $\mathcal{I}$, given by the conjunction of all **invariant** declarations in the module, and

```
1  ghost module toy_protocol {
2    relation voted : node, node
3    relation isleader : node
4    variable quorum : nset.t
5    init ∀n1, n2. ¬voted(n1, n2)
6    init ∀n.¬isleader(n)
7    invariant one_leader = ∀n1, n2. isleader(n1) ∧ isleader(n2) → n1 = n2
8    invariant ∀n, n1, n2. voted(n, n1) ∧ voted(n, n2) → n1 = n2
9    invariant ∀n : node. isleader(n) → (nset.majority(quorum) ∧
10                          ∀n' : node. nset.member(n', quorum) → voted(n', n))
11   use nset.majorities_intersect
12   procedure vote(v : node, n : node) {
13     requires ∀n'.¬voted(v, n')
14     voted(v, n) := true
15   }
16   procedure become_leader(n : node, s : nset.t) {
17     requires nset.majority(s) ∧ ∀n' : node. nset.member(n', s) → voted(n', n)
18     isleader(n) := true
19     quorum := s
20   }
21 }
```

Figure 3. Protocol module for toy leader election.

(4) procedures with pre-post specifications, specified by **requires** and **ensures** declarations (an unspecified condition is *true* by default). The key property of a module is that the verification conditions generated for the invariants and pre-post specifications under the definitions and assumptions are decidable to check (see Section 2.4), which allows predictable automation. We note that our methodology is the first to employ decidable logics for reasoning all the way from protocol design to protocol implementation.

2.3.1 Abstract Protocol Module

Figure 3 lists the module that formalizes the abstract leader election protocol. This is a ghost module, which is only used for the sake of the proof. The module contains two mutable relations that define its state: voted(n_1, n_2) captures the fact that n_1 voted for n_2, and isleader(n) means node n is an elected leader. The initial state of the module specifies that both relations are empty. The state also includes a variable quorum, that remembers the last voting majority observed. The abstract protocol provides a global invariant (denoted by **invariant**), which states that there is at most one leader. This is similar to a class invariant / object invariant in modular reasoning. Next, the module contains a proof that all of its reachable states satisfy the global invariant, by an inductive invariant, and a use of an invariant of the nodeset datatype (included with the **use** declaration in line 11).

The module also provides two procedures that define the abstract protocol steps. Each procedure specifies pre and post conditions. The vote(n_1, n_2) procedure models a vote by n_1 for n_2, and its precondition is that node n_1 has not yet voted. The become_leader(n, s) procedure models the election of n as a leader, and its precondition requires that all nodes in s voted for n, and that s is a majority. Note that

```
1  system module toy_system {
2    message request_vote_msg : node
3    message vote_msg : node, node
4    message leader_msg : node
5    invariant safe = ∀n1, n2. leader_msg(n1) ∧ leader_msg(n2) → n1 = n2
6
7    relation alreadyvoted : node
8    function voters : node → nset.t
9    procedure init(self : node) {
10     alreadyvoted(self) := false
11     voters(self) := nset.emptyset()
12   }
13   procedure request_vote(self : node) {
14     send request_vote_msg(self)
15     if ¬alreadyvoted(self) {
16       alreadyvoted(self) := true
17       voters(self) := nset.add(voters(self, self))
18       ghost { toy_protocol.vote(self, self) }
19     }
20   }
21   procedure cast_vote(self : node, n : node) handles request_vote_msg(n) {
22     if ¬alreadyvoted(self) {
23       alreadyvoted(self) := true
24       send vote_msg(self, n)
25       ghost { toy_protocol.vote(self, n) }
26     }
27   }
28   procedure receive_vote(self : node, n : node) handles vote_msg(n, self) {
29     voters(self) := nset.add(voters(self), n)
30     if nset.majority(voters(self)) {
31       send leader_msg(self)
32       ghost { toy_protocol.become_leader(self, voters(self)) }
33     }
34   }
35   // inductive invariant for the proof:
36   invariant ∀n1, n2. toy_protocol.voted(n1, n2) ↔
37     ((n1 ≠ n2 ∧ vote_msg(n1, n2)) ∨ (n1 = n2 ∧ request_vote_msg(n1)))
38   invariant ∀n1, n2. nset.member(n1,voters(n2)) → toy_protocol.voted(n1,n2)
39   invariant ∀n. leader_msg(n) ↔ toy_protocol.isleader(n)
40   invariant ∀n1, n2. ¬alreadyvoted(n1) → ¬toy_protocol.voted(n1, n2)
41   use toy_protocol.one_leader
42 }
```

Figure 4. System module for toy leader election.

this module is abstract in the sense that it abstracts network communication and uses a global view of the system. In particular, the become_leader does not specify how a node learns that it received a majority of votes.

2.3.2 Concrete Implementation Module

Figure 4 lists the concrete system implementation. We consider systems in which a finite (but unbounded) set of nodes run the same code, and exchange messages. For simplicity, we assume all messages are broadcast to all nodes. Our network model also allows message dropping, duplication and reordering. To define the system implementation, we first define the message types. Lines 2 to 4 define three message types: request_vote_msg that a node uses to propose itself as a leader, vote_msg that a node sends to vote for a candidate, and leader_msg that a node uses to announce it is elected as the leader. The first field of each message is its source node. After defining the message types, we provide the code to

```
1  module nset {
2    type t
3    relation member : node, t
4    relation majority : t
5    function card : t → int
6
7    interpret t as array<int,node>
8    interpret member(n, s) as ∃i.0 ≤ i < array.len(s) ∧ array.value(s, i, n)
9    interpret majority(s) as card(s) + card(s) > card(node.all)
10
11   invariant majorities_intersect = ∀s1, s2. majority(s1) ∧ majority(s2) →
12                 ∃n. member(n, s1) ∧ member(n, s2)
13
14   procedure emptyset() returns s:t {
15     ensures ∀n. ¬member(n, s)
16     s := array.empty()
17   }
18   procedure add(s1 : t, n : node) returns s2 : t {
19     ensures ∀n'. member(n', s2) ↔ (member(n', s1) ∨ n' = n)
20     if member(n, s1) then { s2 := s1 } else { s2 := array.append(s1, n) }
21   }
22   procedure init() {
23     card := λx.0;
24     for 0 ≤ i < array.len(node.all) {
25       invariant ∀s1, s2.(∀n.¬(member(n, s1) ∧ member(n, s2))) →
26         card(s1) + card(s2) ≤ card(node.all)
27       card := λx.((card(x) + 1) if member(array.get(node.all, i), x) else card(x))
28     }
29   }
30 }
```

Figure 5. nset module for node sets, proving the majority intersection property.

be run on each node. This code (lines 7 to 34) defines the local state of each node, as well as procedures that can be executed in response to client requests, or procedures that are message handlers (specified by the **handles** declaration), and executed upon receiving a message from the network. The state components are defined as functions (or relations) whose first argument is a node, so that $f(n)$ denotes the local state of node n. Similarly, all procedures receive as their first argument the `self` identifier of the node that runs them. Since the system module is going to be compiled into an executable code that runs on each node, we syntactically enforce that each node may only access and modify its own local state (via the `self` identifier).

We note that the toy_system module makes use of the nset module to maintain sets of voters, and uses a majority test (line 30). It also makes calls into the ghost module toy_protocol (lines 18, 25 and 32). This allows to establish an invariant that relates the state of the concrete module and the abstract module (lines 36 to 40), and use the proven invariant of the abstract module as a lemma for proving the concrete implementation (line 41).

2.3.3 Nodeset Module

Figure 5 lists the code for the nset module. This module defines a data structure for storing sets of nodes, with operations for adding to a set and testing whether a set is a

majority. To do so, it defines a type t, whose internal interpretation is MDL's built-in array, as declared in line 7. Sets are created using the emptyset and add procedures, which provide a naive implementation of a set, stored as an array of its elements. The module defines the member relation, and provides it with an interpretation via an **interpret** declaration in line 8. As we shall see, this definition creates an executable membership test that is translated to a loop that scans the array.

Most importantly for verifying the leader election modules, the nset module provides the interpreted majority predicate, with the key property that any two majorities intersect. This is stated by the majorities_intersect invariant (line 12). Intuitively, a set is a majority if its cardinality is more than half the total number of nodes. In Figure 5, the majority predicate is interpreted using the card function to compute the cardinality of a set of nodes, and the built-in value node.all, which is an array with the special semantics that it contains all nodes. The card function computes the cardinality of a set of nodes, and it is constructed in the init() procedure of the nset module in a way that establishes the majority intersection property. The proof is by induction, manifested in the loop invariant in line 25.

We note that the loop in init() constructs card via a nesting of N function closures (where N is the number of nodes). This definition of card allows an easy proof of the majority intersection property. A more efficient implementation uses array.len (underlying array length) instead of card to determine if a set is a majority. This implementation is also provable in our system, but requires additional inductive invariants (and ghost state) to prove that card and array.len coincide, and we do not present it here in the interest of simplicity.

2.4 Modular Verification in Decidable Fragments

Recall that each module forms a proof unit. In the sequel, we explain the verification conditions generated for each module, and the theories under which they are checked. In particular, we show how modularity enables decidability. Specifically, in the example we use two decidable fragments of first-order logic: the effectively propositional (EPR) fragment [36] and the finite almost uninterpreted (FAU) fragment [16] (the precise definition of these fragments is not essential for understanding the rest of this section).

Verification conditions Roughly speaking, the verification conditions of a module state that as long as the module's procedures are called in accordance with its preconditions, they will satisfy their postconditions, and the module invariant will be maintained. Technically, for each module, Ivy generates verification conditions (VC's) that check the following properties: 1) The invariant of the module, $\mathcal{I}$, holds at any initial state (i.e., after executing the init() procedure), 2) $\mathcal{I}$ is preserved by every procedure in the module, 3) every procedure in the module satisfies its postcondition, and 4) every procedure in the module respects the preconditions of procedures it calls (from any module). The VC's (2–4) generated for each procedure in the module assume that the execution of the procedure starts from states that satisfy $\mathcal{I}$, the precondition of the procedure, and the assumptions (invariants from other modules). For example, the VC's of each procedure of the toy_protocol module will assume the majorities_intersect invariant of the nset module. Also, calls to (other) procedures are either inlined (i.e., replaced by their body), or summarized (i.e., replaced by their postcondition).

When composing modules, there are additional conditions required for soundness (e.g., that modules do not interfere), which are described in Section 4.

Segregating theories Separating the modules (and their invariants) and breaking the proof into verification conditions for each procedure already plays a big role in the ability to obtain verification conditions that are decidable to check. However, this does not always suffice. Next, we explain how *theories* may either be used to provide interpretations to the state elements of the module, or be abstracted to provide simpler verification conditions that fall into decidable fragments.

Recall that every module also includes type and state declarations and definitions, via the **interpret** declarations. These allow the module to define its relevant *theory*. A theory is a (possibly infinite) set of first-order formulas, which may either be given explicitly (e.g., the theory of total order), or by using a built-in theory (e.g., the theory of linear integer arithmetic). The verification conditions of the module are checked with respect to the provided theory and definitions. Symbols that are given no interpretation in the module are treated as uninterpreted. This may be viewed as a form of abstraction.

Decidable verification conditions in the leader election example In our example, the nset module uses the int type, which introduces the theory of linear integer arithmetic. Furthermore, its majorities_intersect invariant introduces quantifier alternation from nset.t to node. The function voters of the toy_system module introduces a dependency in the opposite direction, as it is a function from node to nset.t. As a result, had the nset module and its invariants been inlined within the proof of toy_system, they would have resulted in verification conditions that use the theory of arithmetic and also contain quantifier alternation cycles, breaking decidability.

In order to break the bad cycle, we do not directly use the majorities_intersect invariant of nset in toy_system. Instead, our proof exploits the toy_protocol module and its one_leader invariant (which does not introduce dependency from node to nset.t). Namely, the invariant one_leader is

assumed when verifying toy_system, and is verified separately as part of the toy_protocol module (assuming the majorities_intersect invariant of nset). In this way, toy_system only contains functions from node to nset.t, while toy_protocol only contains dependencies from nset.t to node, avoiding quantifier alternation cycles.

Verifying each module separately also isolates the reasoning about arithmetic to the nset module. Indeed, the nset module is verified when int is interpreted using the theory of linear integer arithmetic. It also interprets the type nodeset.t as array<int,node> from our built-in array theory (as explained in Section 4, we do not use the theory of arrays, but use the more general FAU). The member and majority relations are interpreted by their definitions (line 8-line 9). The verification conditions for this module are in FAU. When verifying the toy_protocol and toy_system modules, all sorts and relations (including member and majority) are uninterpreted. toy_protocol assumes the majorities_intersect invariant of nset. toy_system assumes the one_leader invariant of toy_protocol and inlines calls to its (ghost) procedures, while summarizing calls to procedures from nset. The resulting verification conditions are in EPR.

We thus see that the separation between the three modules allows us to obtain decidable verification conditions.

2.5 Compiling to C++ and Runtime System

In order to obtain a verified implementation, Ivy generates C++ code. During this phase, ghost code (the ghost module toy_protocol in our example) is sliced out, and every call to a procedure of a ghost module is treated as skip. The remaining code is translated to C++, as detailed next.

Translation of Primitive Language Constructs Every procedure is translated to a C++ function in a straightforward syntax-directed manner. Control flow constructs are translated into the corresponding C++ constructs. Interpreted sorts are given appropriate representations. The built-in type int is represented by machine integers, arrays are represented by the STL `std::vector` template, and record types are represented by C++ `struct`.[1] Variables of function sort are represented by function closures equipped with a memo table. Every type in a non-ghost module must have one of the above as its interpretation.

Ivy code may contain quantified formulas as control flow conditions (e.g., as the condition of an if statement). In non-ghost context, these must be of the form $\exists i : \text{int}.\ a \le i \le b \wedge \varphi(i)$ or $\forall i : \text{int}.\ a \le i \le b \rightarrow \varphi(i)$. Ivy translates such conditions into for loops. In the Toy Leader Election example, this mechanism is used to compile the definition of member (Figure 5 line 8) to an executable test.

[1] In the current implementation, integer overflow is not addressed. We intend this to be handled by an efficient bignum package.

Network & Runtime The generated C++ code is intended to operate in a distributed setting, where each node runs the same program, and nodes communicate via message passing (for simplicity, we assume messages are broadcast to all nodes). Accordingly, a system module must define the message types, and the local state and procedures of each node. The local state relations and functions, and the local procedures, all have an argument of the built-in type node as their first parameter, which represents the local node. The generated C++ code includes variables of the appropriate type that represent the local state of the node. It also includes the local procedures which can only access the local state of the node, and may also send messages. Some procedures are designated as message handlers.

The generated C++ code is linked to client code to form the complete application. The client code may call the generated procedures (such as request_vote in the example) in order to use the service provided by the verified code.

The generated code also includes an additional shim that takes care of sending and receiving messages, and firing timers. Namely, message sending is translated to calls to appropriate shim functions, and the shim calls message handlers or timeout handlers when messages are received or timers expire, respectively. The shim also initializes the values of `self` and node.all with a node identifier, and an array of all node identifiers, respectively. This information is obtained at run time from a configuration file or command line arguments. The user is trusted to properly configure the system, i.e., to run processes with unique id's, and to provide each process with a correct list of all other process id's and network information (e.g., IP addresses and ports).

3 Preliminaries

In this section we provide a brief background on first-order logic, theories and the decidable fragments used in this work.

3.1 Formulas and theories

We consider many-sorted first-order logic with equality, where formulas are defined over a set $\mathcal{S}$ of first-order sorts, and a vocabulary Σ which consists of sorted constant symbols and function symbols. Constants have first-order sorts in $\mathcal{S}$, while functions have sorts of the form $(\sigma_1 \times \cdots \sigma_n) \rightarrow \tau$, where $\sigma_i, \tau \in \mathcal{S}$. In other words, functions may not be higher-order. We assume that $\mathcal{S}$ contains a sort $\mathbb{B}$ that is the sort of propositions. A function whose range is $\mathbb{B}$ is called a relation. If s is a symbol or term and t is a sort, then $s : t$ represents the constraint that s has sort t. We elide sort constraints when they can be inferred. A Σ-structure maps each sort $t \in \mathcal{S}$ to a non-empty set called the *universe* of t, and each symbol $s : t$ in Σ to a value of sort t.

For a set of formulas T, we denote by $\Sigma(T)$ the vocabulary of T, that is, the subset of Σ that occurs in T. A *theory* T is a (possibly infinite) set of formulas. We use theories to give

concrete interpretations of the symbols in Σ. For example, a given sort might satisfy the theory of linear order, or the theory of arithmetic. In particular:

Definition 3.1. A theory T is $\mathcal{V}$*-conservative*, where $\mathcal{V} \subset \Sigma$, if every $(\Sigma \setminus \mathcal{V})$-structure σ can be extended to a Σ-structure σ' such that $\sigma' \models T$.

Intuitively speaking, $\mathcal{V}$-conservativeness means that T defines the symbols in $\mathcal{V}$, possibly in terms of other symbols. We will also say that a $\mathcal{V}$-conservative theory *interprets* the symbols in $\mathcal{V}$. We can compose conservative theories to obtain conservative theories:

Theorem 3.2. *If T is a $\mathcal{V}$-conservative theory and T' is a $\mathcal{V}'$-conservative theory, where $\Sigma(T)$ and $\mathcal{V}'$ are disjoint, then $T \cup T'$ is a $(\mathcal{V} \cup \mathcal{V}')$-conservative theory.*

That is, definitions can be combined sequentially, provided earlier definitions do not depend on later definitions.

3.2 Decidable fragments

We consider fragments of first-order logic for which checking *satisfiability*, or *satisfiability modulo a theory* (i.e., satisfiability restricted to models of a given theory T), is decidable.

Effectively Propositional Logic (EPR) The effectively propositional (EPR) fragment of first-order logic, also known as the Bernays-Schönfinkel-Ramsey class is restricted to first-order formulas with a quantifier prefix $\exists^*\forall^*$ in prenex normal form defined over a vocabulary Σ that contains only constant and relation symbols, and where all sorts and symbols are uninterpreted. Satisfiability of EPR formulas is decidable [26]. Moreover, formulas in this fragment enjoy the *finite model property*, meaning that a satisfiable formula is guaranteed to have a finite model.

A straightforward extension of this fragment allows *stratified* function symbols and quantifier alternation, as formalized next. The *Skolem normal form* of a formula is an equisatisfiable formula in $\forall^*$ prenex normal form that is obtained by converting all existential quantifiers to Skolem functions. The *function graph* of a formula is a graph whose vertex set is $\mathcal{S} \setminus \{\mathbb{B}\}$, having an edge (s,t) if there is a function symbol occurring in the formula's Skolem normal form with s in its domain and t in its range. Notice this means that a formula of the form $\forall x : s.\ \exists y : t.\ \varphi$ has an edge (s,t) in its function graph, since the Skolem function for y is of sort $s \to t$. A *bad cycle* in the function graph of φ is one containing a sort s such that some variable of sort s is universally quantified in the Skolem normal form of φ.

A formula is *stratified* if its function graph has no bad cycles. Notice that all EPR formulas are stratified (since all the Skolem symbols are constants) and so are all the quantifier-free formulas. A formula φ is *virtually stratified* if there is *any* consistent assignment of sorts to symbols in $\Sigma(\varphi)$ under which φ is stratified.

As an example, the formula $\forall x : s.\ \exists y : t.\ f(x) = y$ is stratified, since the function graph contains only the edge (s,t). On the other hand, $\forall x : s.\ \exists y : t.\ f(y) = x$ is not stratified, because the Skolem function for y has sort $s \to t$, while f has sort $t \to s$. The formula $\forall x : s.\ f(g(x) : t) = y : s$ is not stratified, since g has sort $s \to t$ while f has sort $t \to s$. However, it *is* virtually stratified, since we can resort it as $\forall x : s.\ f(g(x) : t) = y : u$. Also, notice that $\forall x : s.\ f(g(x) : t) = y : t$ is stratified even though there is a cycle containing sort t, because this cycle does not contain a universally quantified variable.

The extended EPR fragment consists of all virtually stratified formulas. The extension maintains both the finite model property and the decidability of the satisfiability problem (this is a special case of Proposition 2 in [16]).

Finite Almost Uninterpreted Fragment (FAU) Formulas in the almost uninterpreted fragment [16] are defined over a vocabulary that consists of the usual interpreted symbols of Linear Integer Arithmetic (LIA), equality and bit-vectors, extended with uninterpreted constant, function and relation symbols. In this work, we will not use bit-vectors. We recall that LIA includes constant symbols (e.g., $1, 2, \ldots$), function symbols of linear arithmetic (e.g., "+", but not multiplication), and relation symbols (e.g., "$\leq$"), all of which are interpreted by the theory, which includes all formulas over this vocabulary that are satisfied by the integers. A formula (over the extended vocabulary) is in the *essentially uninterpreted* fragment if variables are restricted to appear as arguments to uninterpreted function or relation symbols. The *almost uninterpreted* fragment also allows variables to appear in inequalities in a restricted way (for example, inequalities between a variable and a ground term are allowed). For example $\forall x : \text{int}.\ x + y \leq z$, is not in the fragment, since the variable x appears under the interpreted relation $\leq$. However $\forall x : \text{int}.\ f(x) + y \leq z$ is allowed, as is $\forall x : \text{int}.\ x \leq y$.

The *finite* almost interpreted fragment (FAU) is defined as the set of almost interpreted formulas that are stratified as defined in [16]. Satisfiability of FAU modulo the theory is decidable. In particular, in [16] a set of groundings is defined that is sufficient for completeness. In FAU, this set is finite, which implies decidability. Moreover, it implies that every satisfiable formula has a model in which the universes of the uninterpreted sorts are finite. This is useful for providing counterexamples. The FAU fragment also subsumes the array property fragment described in [5].

Of particular importance for our purposes, the SMT solver Z3 [11] is a decision procedure for the FAU fragment. This is because its model-based quantifier instantiation procedure guarantees to eventually generate every grounding in the required set. This gives us a rich language in which to express our verification conditions, without sacrificing decidabilty. The subject of this paper is how to structure the proofs

of implementations of distributed protocols such that the verification conditions fall into this fragment.

4 Modular Proofs

In this section we describe our modular reasoning system, using a very simple procedural language as a model of MDL. We will point out along the way the correspondence of features of MDL to the model language.

4.1 A model language

The statements in our model language are defined as follows:

Definition 4.1. Let $\mathcal{N}$ be a set of *procedure names* and $\mathcal{V}_P \subset \Sigma$ a set of program variables. The *program statements* $\mathcal{S}$ are defined by the following grammar:

$$\begin{aligned}\mathcal{S} \quad ::= \quad & c : \tau := t : \tau \mid (\mathcal{S}; \mathcal{S}) \mid \text{while } p\ \mathcal{S} \\ & \mid \text{if } p\ \mathcal{S}\ \mathcal{S} \mid \text{call } N \mid \text{skip}\end{aligned}$$

where c is in $\mathcal{V}_P$, t a term over Σ, τ a sort, and $p \in \mathcal{P}$.

The mutable program variables are a subset of the logical symbols. The statements have the expected semantics. For now, only first-order program variables c can be assigned, since a term t can only have a first-order type. We will relax this restriction in Section 4.5.

The *Hoare triples* $\mathcal{H}$ are denoted $\{\varphi\}\ \sigma\ \{\psi\}$, where $\varphi, \psi \in \mathcal{P}$ and $\sigma \in \mathcal{S}_{\mathcal{N}}$. Our notion of procedure definition is captured by the following definition:

Definition 4.2. A *context* is a partial function from $\mathcal{N}$ to $\mathcal{H}$. A context is denoted by a comma-separated list of *procedure definitions* of the form $n{:=}H$, where $n \in \mathcal{N}$ and $H \in \mathcal{H}$, such that the names n are unique.

Intuitively, a context is a collection of procedure definitions with corresponding pre/post specifications. In MDL, the precondition φ of a procedure is introduced by the "requires" keyword and the postcondition by "ensures".

We write $P \vdash_T \{\varphi\}\sigma\{\psi\}$ to denote the judgment that, *assuming* context P and background theory T, if σ terminates then it terminates in a T-model satisfying ψ. In derivation rules, we will drop the theory T if it is the same for all judgments.

The *derivable* judgments are defined by the standard rules of Hoare logics (see Appendix). To these, we add the Assumption rule, which allows us to use the assumption that a procedure satisfies its specification at a call site:

$$\frac{}{n{:=}\{\varphi\}\ \sigma\ \{\psi\}, P \vdash \{\varphi\}\ \text{call } n\ \{\psi\}}$$

To discharge such an assuption, we add the Inline rule, as follows:

$$\frac{P \vdash \{\varphi\}\ \sigma\ \{\psi\}}{n{:=}\{\varphi'\}\ \sigma\ \{\psi'\}, P \vdash \{\varphi \wedge \varphi'\}\ \text{call } n\ \{\psi \wedge \psi'\}}$$

That is, any fact that can be proved about the body of procedure n in a given context can be used at a call site of n. Notice that we must still satisfy any specified pre-condition φ' and may use the specified post-condition ψ'. In effect, this allows us to inline a procedure definition at a call site. This is relatively complete for non-recursive programs, which include the examples to treat here.

4.2 Modules and the assume/guarantee rule

A *module* is a procedural program that exports procedure definitions to its environment and has a determined set of initial states. In the sequel, if f is a partial function, we will write $\text{pre}(f)$ for its pre-image and $\text{img}(f)$ for its image. If P is a context, we will write $\text{called}(P)$ for the set of names n such that "call n" occurs in P.

Definition 4.3. A *module* is a triple (P, E, I), where:

- P is a context.
- $E \subseteq \text{pre}(P)$ is the set of *exports*.
- $I \in \mathcal{P}$ gives the *initial state* of the module.

The module is said to be *semi-closed* if $\text{called}(P) \subseteq \text{pre}(P)$.

That is, P gives a set of procedure definitions with pre/post specifications and E gives the subset of these definitions that is exported to the environment. A semi-closed module calls only procedures it defines. We will write P_M, E_M, I_M, respectively, for the components of module M.

We will write $P \vdash M$ to represent the judgment that, in context P, in a non-interfering environment, module M satisfies its pre/post specifications. The axiomatic semantics of modules is given by the following rule:[2]

$$\frac{\begin{array}{l} I_M \models \mathcal{I} \\ \text{for } n{:=}\{\varphi\}\ \sigma\ \{\psi\} \text{ in } P_M{:} \\ \quad \begin{cases} P, P_M \vdash \{\varphi\}\ \sigma\ \{\psi\} & \text{if } n \notin E \\ P, P_M \vdash \{\mathcal{I} \wedge \varphi\}\ \sigma\ \{\mathcal{I} \wedge \psi\} & \text{else} \end{cases} \end{array}}{P \vdash M}$$

In this rule, $\mathcal{I}$ is an inductive invariant that holds between calls to exported procedures. It must hold in the initial states and be preserved by all exported procedures.[3] In MDL, this invariant $\mathcal{I}$ is given by the conjunction of the "invariant" declarations in the module.

If $M = (P, E, I)$ is a module and $E' \subseteq E$, the *restriction* of M to E', denoted $M \downarrow E'$, is (P, E', I). The *refset* $\text{ref}(M)$ of M is the subset of $\mathcal{V}_P$ occurring in $\text{img}(P)$ or I. The *modset* $\text{mod}(M)$ of M is the subset of $\mathcal{V}_P$ *assigned* in P or occurring in I.

Definition 4.4. Module M is said to *interfere* with module M', denoted $M \triangleright M'$, if $\text{mod}(M) \cap \text{ref}(M') \neq \emptyset$ or if $\text{called}(P_M) \cap \text{pre}(P_{M'}) \not\subseteq E_{M'}$.

[2] This rule is incomplete if the module calls its own exported procedures.

[3] It might seem necessary that the environment not modify any variable in $\Sigma(\mathcal{I})$ but this is not needed. One way to see this is that a variable in $\mathcal{I}$ not referenced in M can be renamed while preserving the proof.

In other words, M interferes with M' if it either modifies a variable that M' references, or if it calls an internal procedure of M'. Module M is said to be *compatible* with module M' if $\text{pre}(P_M) \cap \text{pre}(P_{M'}) = \emptyset$ and $\Sigma(I_M) \cap \Sigma(I_{M'}) \cap \mathcal{V}_P = \emptyset$.

Definition 4.5. If $M = (P, E, I)$ and $M' = (P', E', I')$ are compatible modules, the *composition* of M' and M, denoted $M' + M$, is $(P \cup P', E \cup E', I \wedge I')$

For simplicity, we will restrict our attention to *layered* compositions. This is a composition $M + M'$ in which M is semi-closed, M' does not interfere with M, and the composition $M + M'$ is itself semi-closed (that is, all external calls of M' are satisfied by M). We can think of this as the case when service M' is layered on top of service M.

Theorem 4.6. *The following inference rule can be derived:*

$$\frac{\begin{array}{c}\vdash M \\ P_M \vdash M'\end{array}}{\vdash (M + M') \downarrow E_{M'}} M' \not\vdash M$$

That is, to verify M' layered on M, we first verify M in an empty context, then verify M' using the proved specifications of the exported procedures of M. Intuitively, this works because M' cannot violate the invariant of M by merely calling its exported procedures. Therefore the specifications of M that were proved in isolation also hold in the composition. This simple rule is very restrictive, since it requires hiding all of the exports of M. We can extend it to allow exporting procedures from M as well, under the condition that they preserve the invariant of M'.

When proving the second premise of this rule, we have two options. We can apply the Assumption rule at the procedure calls of M', and thus use *only* the specifications of M'. Or, we can apply the Inline rule, in which case we use both the proved specifications and the procedure body. Both of these options are useful. In Toy Leader Election we applied the first approach to the abstract datatype for sets, and the second approach to the abstract protocol model. This allowed us to write an invariant relating the states of the two models.

The Layer rule gives us no direct access to invariants that may have been proved about the lower module M. If an invariant φ of M is needed, we can add an empty procedure to M with post-condition φ and call it where needed in M'. In MDL the "use" keyword indicates an invariant of another module that should be used in this way. The given invariant of the sub-module will be used at entry to every exported procedure, and after every call to a procedure of the sub-module.

4.3 Ghost modules and slicing

Definition 4.7. If P is a context, the *slice* of P, denoted $\text{slice}(P)$ is the set of procedure definitions which contains $n{:=}\{\varphi\}\text{skip}\{\psi\}$ for each $n{:=}\{\varphi\}\sigma\{\psi\}$ in P. If M is a module, $\text{slice}(M)$ denotes $(\text{slice}(P_M), E_M, I_M)$.

The following derived rule can be used to slice out a "ghost" module that is used only for the purpose of the proof, provided the ghost module does not interfere:

Theorem 4.8. *The following inference rule can be derived:*

$$\frac{\vdash (M + M') \downarrow E_{M'}}{\vdash (\text{slice}(M) + M') \downarrow E_{M'}} M \not\vdash M' *$$

The side condition (*) of this rule requires that all procedures in P_M must terminate starting in all states satisfying their preconditions. For the examples presented here, it suffices to verify that P_M is not recursive and contains no "while" statements. Also, it requires that the initial condition I_M be $\mathcal{V}_P$-conservative. In other words, every model of the theory must have an extension to the program variables satisfying I_M. In practice we must prove this using Theorem 3.2, which means that I_M must be a conjunction of a sequence of definitions.

The ghost module M can be used like a lemma in the proof of M'. That is, we make use of its post-conditions and then discard it, as we did with the abstract protocol model in Toy Leader Election.

4.4 Segregating theories

To allow us to segregate theories, we add one derived rule Theory to our system:

Theorem 4.9. *The following inference rule can be derived:*

$$\frac{\begin{array}{c}T, T' \models T'' \\ P \vdash_{T \cup T''} \{\varphi\}\ \sigma\{\psi\}\end{array}}{P \vdash_{T \cup T'} \{\varphi\}\ \sigma\{\psi\}}$$

In other words, what can be proved in a weaker theory can be proved in a stronger theory. This allows us, for example, to replace the theory of arithmetic with the theory of linear order, or to drop function definitions that are not needed in a given module. In Toy Leader Election, for example, we dropped the theory LIA and the definition of `majority` when verifying the abstract model and the implementation, but used them when verifying `nset`.

4.5 Language extensions

In this section we introduce some useful extensions to the basic language which, while straightforward, cannot be detailed here due to space considerations.

Though we have modeled procedure calls as having no parameters, it is straightforward to extend the language to include call-by-value with return parameters. In the following we assume such an extension.

We allow assignments of the form $f{:=}\lambda x.\ e$, where f is function, since the resulting verification conditions can still be expressed in first order logic [35]. In the compiled code, the resulting value of f is a function closure. The assignment $f(a){:=}e$ is a shorthand for $f{:=}\lambda x.$ if $x = a$ then e else $f(x)$.

We provide built-in theories for integers (with the usual arithmetic operators), bit vectors (also with arithmetic) and finite immutable arrays (with functions for length, select, update and element append). For each finite sort σ (such as node in Toy Leader Election) the array theory provides a constant σ.all that contains all elements of sort σ. We used this feature to define the notion of a majority of nodes in Toy Leader Election. Our built-in theories are encoded in FAU.

A module M may have a special initialization procedure M.init that is called by the environment before any exports. This procedure must establish the module invariant with no precondition, as it does, for example, in module `nset`.

4.6 Modeling network communication

For simplicity, we will introduce only a model of a broadcast datagram service, as used in Toy Leader Election. Other services can be modeled similarly. For each sort $\mathcal{M}$ of mesages transmitted on te network, we introduce an abstract relation sent($m : \mathcal{M}$) to represent the fact that message m has been broadcast in the past. We add to the language a primitive "send $m : \mathcal{M}$" whose semantics is defined by the following rule:

$$\frac{P \vdash \{\varphi\}\ \text{sent}(m){:=}\text{true}\ \{\psi\}}{P \vdash \{\varphi\}\ \text{send}\ m\ \{\psi\}}$$

That is, the effect of "send $m : \mathcal{M}$" is to add message m to the set of broadcast messages of sort $\mathcal{M}$. A module using network services for sort $\mathcal{M}$ exports a procedure "recv$_{\mathcal{M}}$" that is called by the network. This procedure takes two parameters: p : pid to represent the receiving process id and $m : \mathcal{M}$ to represent the received message. We use $\vdash^{\mathcal{N}} M$, where $\mathcal{N}$ is a collection of sorts, to represent the judgment that M satisfies its specifications when composed with a network that handles mesages of sorts in $\mathcal{N}$. This judgment can be derived by the following Network rule:

$$\frac{\begin{array}{c}\text{sent}(m : \mathcal{M}) \models_T \varphi \\ \vdash_T^{\mathcal{N}} M\end{array}}{\vdash_T^{\mathcal{N},\mathcal{M}} M \downarrow (\mathcal{N} \setminus \text{recv}_{\mathcal{M}})} \quad \begin{array}{l}\text{recv in } E_M \\ \text{recv } := \{\varphi\}\ \sigma\ \{\psi\} \text{ in } P_M\end{array}$$

In other words, we can assume that the system calls recv(p, m) only with messages m that have already been broadcast. This yields a very weak network semantics, allowing messages to be dropped, reordered and duplicated. In MDL, we used the keyword "handles" to indicate which procedures are used to handle received messages of a given sort. The keyword "system" indicates a top-level module, to which the above rule should be applied.

4.7 Concurrency and parametricity

Thus far, we have considered a purely sequential program that presents exported procedures to be called by its environment and assumes that each call terminates before the next call begins. This semantics is implicit in Definition 4.3. In reality, however, calls with different values of the process id parameter p will be executed concurrently. We need to be able to infer that every concurrent execution is sequentially consistent, that is, it is equivalent to some sequential execution when only the local histories of actions are observed. To do this, we use Lipton's theory of left-movers and right-movers [27], in much the same way as is done in the IronFleet project [17]. Since this argument does not relate directly to the use of decidable theories, we only sketch it here.

First, we need to show that any two statements executed by two different processes, excepting "send" statements, are independent. To do this, we require that every exported procedure have a first parameter p : pid (representing the process id). We verify statically that every program variable reference (after slicing the ghost modules) is of the form $f(p, \ldots)$ where p is the process id parameter. Another way to say this is that all statements except send statements are "both-movers" in Lipton's terminology. Moreover, by construction, every call from the environment consists of an optional message receive operation, followed by a combination of both-movers and sends. Since receive is a right-mover and send is a left-mover, it follows that every call can be compressed to an atomic operation, thus the system is sequentially consistent by construction.

When we compile a module to executable code, we take the parameter p as a fixed value given at initialization of the process. We use this constant value to partially evaluate all program variable references, thus allowing the compiled code to store the only the state of one process.

4.8 Verification conditions

We use the inference rules above to generate verification conditions (VC's) in the usual manner, as in tools such as ESC Java [15] and Dafny [25]. These are validity checks that result from the side conditions of the Module, Network, and Theory rules, and the standard rule of consequence. The verifier checks that each VC is in one of our decidable fragments (taking into account any built-in theories used) and issues a warning if it is not. The warning may exhibit, for example, a bad cycle in the function graph. In case a VC is determined to be invalid by the Z3 prover, the counter-model produced by Z3 is used to create a program execution trace that demonstrates the failure of the VC.

5 Evaluation

To evaluate our methodology, we applied it to develop verified implementations of Raft [33] and Multi-Paxos [23]. We used the Ivy [35] system, which uses Z3 [11] to check verification conditions. Both Raft and Multi-Paxos implement a centralized shared log abstraction, i.e. a write-once map from indices to values. This abstraction can be used to implement a distributed fault-tolerant service, using the state machine replication approach [37].

5.1 Raft and Multi-Paxos

Raft The Raft protocol operates in a sequence of *terms*. In each term a leader is elected in a way that is similar to the toy leader election protocol presented in Section 2, and the leader then replicates its log on the other nodes. In this work, we do not consider log truncation and crash recovery.

Our verified Raft implementation follows the modularity principles presented in Section 2. It consists of an abstract protocol ghost module and its correctness proof, a module for the system implementation, and two datatype modules for node sets with majority testing logs represented as built-in arrays. The implementation module is fairly straightforward, and similar to an implementation in an imperative C-like language. It contains function symbols that map node to various sorts that capture the node state.

The ghost module contains the core of the proof, and is where most of the verification effort was spent. It contains auxiliary relations, used both to avoid quantifier alternation cycles and to record information used in the inductive invariant. For example, while the concrete implementation contains a function $t : \text{node} \to \text{term}$, the ghost module abstracts it as a relation $t_g : \text{node}, \text{term}$, intended to capture $t_g(n, x) \equiv t(n) = x$. This avoids quantifier alternation from node to term.

Verification of the abstract protocol ghost module is done in EPR, with log indices and terms abstracted as linear orders. This module provides the key correctness invariant, i.e., consistency and persistence of committed entries. The verification of the implementation module is also done in EPR. However, the abstract protocol and the implementation use different quantifier alternation stratification orders, so a non-modular proof would lead to undecidability. The modules for nodesets and logs are verified in the FAU fragment.

Multi-Paxos Our approach to implementing Multi-Paxos and the features implemented are largely similar to that of Raft. One interesting feature of the Multi-Paxos proof is that the protocol design contains a procedure whose precondition asserts that for every index in the log, there exists a node n such that some condition holds. This quantifier alternation is stratified in the protocol design but not in the implementation (which calls the procedure).

To satisfy the procedure's precondition without breaking stratification in the implementation, we create a ghost module, called witness, specifying an uninterpreted type that we can think of as a map from index to node, and we pass a parameter of this type to the protocol-design procedure. The witness module has two levels of specification: the first level avoids stratification problems by abstractly specifying the map as a relation, while the second level, hidden for the implementation proof, enables to instantiate the problematic quantifier by defining the map in terms of a function.

5.2 Verification Effort

The Raft verification took 2 person-months, including 1.5 person-months for verifying the abstract protocol, and 2 weeks for verifying the implementation. The code contains 560 SLOC of implementation and 200 SLOC of invariants and ghost code, for a proof-to-code ratio of 0.36. Obtaining the Multi-Paxos implementation from the abstract protocol (which was already proved) took approximately one week of work, reusing some of the work done for Raft. The code consists of 330 SLOC of implementation and 266 SLOC of invariants and ghost code (including the witness module). In total, the proof to code ratio is of about 0.8 (counting invariants and ghost code as proof). Ivy successfully discharges all VCs of both Multi-Paxos and Raft in a few minutes on a conventional laptop. During the development, Ivy quickly produced counterexamples to induction and displayed them graphically, which greatly assisted in the verification process.

For comparison, IronFleet's IronRSL [17] implementation is part of a verification effort of 3.7 person-year (including another verified protocol), and VC checking was outsourced to cloud machines to obtain verification times acceptable for interactive use. IronRSL consists of roughly 3000 implementation SLOC and 12000 proof SLOC (excluding generic IronFleet libraries). This gives a proof/code ratio of about 4. However, as evidenced by its larger code-base, IronRSL has more features than the Ivy implementations presented, such as log truncation, batching, and state-transfer. The Verdi proof of the Raft protocol [41] consists of 50000 lines of proof for 530 lines of implementation and required several person-years. The proof is done in the Coq [3] proof assistant and has a smaller trusted code base than Ivy.

5.3 Verified System's Performance

To evaluate the performance of our verified systems, we developed a key-value store that replicates data across multiple nodes using either Raft or Multi-Paxos. The implementations extracted from Ivy handle all internal communication, while the (unverified) key-value component handles communication with clients and data storage; it consists of 785 SLOC of C++. We benchmarked the performance of these systems against that of `vard` [40], a verified key-value store, as well as `etcd` [10], an unverified, production-quality key-value store. Both `vard` and `etcd` use Raft for consensus.

We benchmarked all systems on a cluster of three Amazon EC2 `t2.small` nodes, each with 1 CPU core and 2GB of RAM. We then started 50 closed-loop client threads on a fourth `t2.small` node. The threads sent 10000 requests (a randomized 50/50 mix of `GET`s and `PUT`s). `vard` and `etcd` both persist data to disk, while our key-value store does not; therefore we modified the `vard` shim to stop it from writing to disk, and we configured `etcd` to use a RAM-disk.

Our results are summarized in Table 1. Both our verified key-value stores achieve similar performance to `vard`, while

Table 1. Throughput (requests/s) and average request latency (ms) of gets and puts to the key-value stores.

System	Throughput	Get	Put
Ivy-Raft	550.1	84.4	85.1
Ivy-Multi-Paxos	554.2	83.9	84.0
`vard`	624.2	79.8	80.0
`etcd`	2943.5	16.5	17.1

all three verified stores are roughly 5× slower than `etcd`. We have not significantly optimized our implementations, so it is unsurprising that a production-quality system achieves better performance.

6 Related Work

Fully Automatic Verification Fully automatic push-button verification is usually beyond reach because of undecidability. Bounded checking is successfully used in systems like Alloy [19] and TLA+ [24] to check correctness of designs up to certain number of nodes. This is useful, due to the observation that most bugs occur with small number of nodes. However as observed by Amazon [31] and others it is hard to scale these methods even for very few, e.g., 3 nodes. Also many of the interesting bugs occur in the implementation. One interesting approach to obtain decidability of fully automatic verification is via limiting the class of programs [4, 20–22]). Another approach is to use sound and incomplete procedures for deduction and invariant search for logics that combine quantifiers and set cardinalities [39]. However, realistic distibuted systems as the ones we consider are beyond reach of such this technique. Another direction, explored in [2], is to verify limited properties (e.g., for absence of deadlocks), using a sound but incomplete decidable check. None of the state-of-the-art techniques for fully automatic verification can prove properties such as consistency for systems implementations. Moreover, when automatic methods fail, the user is struck without any solution.

Interactive Verification Recent projects such as Verdi [40] and IronFleet [17] demonstrate that distributed systems can be proved all the way from the design to an implementation. The DistAlgo project [6, 28] develops programming methodologies for interactively verifying distributed systems, based on the TLA+ proof system [7]. However, interactive verification requires tremendous human efforts and skills and thus has limited success so far. Our work can be considered as an attempt to understand how much automation is achievable using modularity. We argue that invariants provide a reasonable way to interact with verification systems since one does not need to understand how decision procedures such as SMT works. In this work we express invariants in first-order logic. In the future, it may be possible to develop high-level specification approaches.

Decidable Reasoning about Distributed Protocols PSync [13] is a DSL for distributed systems, with decidable invariant checking in the $\mathbb{CL}$ logic [12]. Decidability is obtained by restricting to the partially synchronous Heard-Of Model. A partially synchronous model is also used in [1], which develops a decidable fragment that allows some arithmetic with function symbols and cardinality constraints. Compared to these works, our approach considers a more general setting of asynchronous communication, and uses more restricted and mainstream decidable fragments that are supported by existing theorem provers. This is enabled by our use of modularity. Our approach of applying modularity to obtain decidability will benefit and become more powerful as more expressive decidable logics are developed and supported by efficient solvers.

Recently, [34] showed how to verify abstract protocols of the Paxos family in EPR, and [29] used it for cache coherence. In this work we use EPR to reason about abstract protocols but also go beyond the protocol level to verify an actual efficient system implementation. The gap between the abstract protocol and the system implementation presents a major technical challenge. We bridge this gap by the novel use of modularity to obtain decidability, as well as the use of the FAU fragment [16], which is more expressive than EPR.

Modularity in Verification The utility of modularity for simplified reasoning was already recognized in the seminal works of Hoare and Dijkstra (e.g., [18]). Proof assistants such as Isabelle/HOL [32] and Coq [3] provide various modularity mechanisms. Indeed, existing deductive verification engines such as Dafny [25] employ modularity to simplify reasoning a way that is similar to ours. In this landscape, our chief novelties are the use of modularity for decidability, and a methodology for modular, decidable reasoning about distributed systems. We note that unlike Dafny, Ivy performs syntactic checks to ensure that verification conditions are in decidable logics. Thus, the user either receives an error message and corrects the specification or can be assured that the verification problem is solvable. Our evaluation demonstrates that out methodology is useful for verifying realistic systems, and that it drastically reduces the verification effort.

7 Conclusion

Modularity is well recognized as a key to scalability of systems. This paper shows that modularity enables decidability of reasoning about real implementations of distributed protocols. At the implementation level, distributed protocols involve arithmetic, unbounded sets of process and unbounded data structures. For this reason, we might expect that reasoning about these systems would require the use of undecidable logics. We have seen however, that by a fairly simple modular decomposition, we can separate the proof into lemmas that reside in decidable fragments, which in turn can make the use of automated provers more predictable and transparent.

References

[1] Francesco Alberti, Silvio Ghilardi, and Elena Pagani. 2016. Counting Constraints in Flat Array Fragments. In *Automated Reasoning*. Springer, Cham, 65–81.

[2] Alexander Bakst, Klaus von Gleissenthall, Rami Gökhan Kici, and Ranjit Jhala. 2017. Verifying distributed programs via canonical sequentialization. *PACMPL* 1, OOPSLA (2017), 110:1–110:27. https://doi.org/10.1145/3133934

[3] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer. https://doi.org/10.1007/978-3-662-07964-5

[4] Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. 2015. *Decidability of Parameterized Verification*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00658ED1V01Y201508DCT013

[5] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2006. What's Decidable About Arrays?. In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*. 427–442. https://doi.org/10.1007/11609773_28

[6] Saksham Chand, Yanhong A. Liu, and Scott D. Stoller. 2016. Formal Verification of Multi-Paxos for Distributed Consensus. In *FM 2016: Formal Methods: 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings 21*. Springer, 119–136.

[7] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. 2010. The TLA+Proof System: Building a Heterogeneous Verification Platform. In *Proceedings of the 7th International Colloquium Conference on Theoretical Aspects of Computing (ICTAC'10)*. Springer-Verlag, 44–44.

[8] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay Ileri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. 270–286.

[9] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. 18–37. https://doi.org/10.1145/2815400.2815402

[10] CoreOS 2014. etcd: A highly-available key value store for shared configuration and service discovery. (2014). https://github.com/coreos/etcd.

[11] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science)*, Vol. 4963. Springer, 337–340.

[12] Cezara Dragoi, 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.

[13] Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. 2016. PSync: A Partially Synchronous Language for Fault-Tolerant Distributed Algorithms. *ACM SIGPLAN Notices* 51, 1 (2016), 400–415.

[14] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. 287–305.

[15] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. ACM, 234–245. https://doi.org/10.1145/512529.512558

[16] Yeting Ge and Leonardo De Moura. 2009. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *International Conference on Computer Aided Verification*. Springer, 306–320.

[17] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP*. 1–17.

[18] C. A. R. Hoare. 1972. Proof of correctness of data representations. 1, 4 (1972), 271–281.

[19] Daniel Jackson. 2006. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.

[20] Igor Konnov, Marijana Lazic, Helmut Veith, and Josef Widder. 2017. A Short Counterexample Property for Safety and Liveness Verification of Fault-Tolerant Distributed Algorithms. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, 719–734.

[21] Igor Konnov, Helmut Veith, and Josef Widder. 2015. SMT and POR Beat Counter Abstraction: Parameterized Model Checking of Threshold-Based Distributed Algorithms. In *Computer Aided Verification*. Springer, Cham, 85–102.

[22] Igor V. Konnov, Helmut Veith, and Josef Widder. 2015. What You Always Wanted to Know About Model Checking of Fault-Tolerant Distributed Algorithms. In *Perspectives of System Informatics - 10th International Andrei Ershov Informatics Conference, PSI 2015, in Memory of Helmut Veith, Kazan and Innopolis, Russia, August 24-27, 2015, Revised Selected Papers (Lecture Notes in Computer Science)*, Manuel Mazzara and Andrei Voronkov (Eds.), Vol. 9609. Springer, 6–21. https://doi.org/10.1007/978-3-319-41579-6_2

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

[24] Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[25] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer, 348–370.

[26] Harry R. Lewis. 1980. Complexity results for classes of quantificational formulas. *J. Comput. System Sci.* 21, 3 (1980), 317 – 353.

[27] R. J. Lipton. 1975. Reduction: A method of proving properties of parallel programs. *Commun. ACM* 18, 12 (1975), 717–âĂŞ721.

[28] Yanhong A. Liu, Scott D. Stoller, and Bo Lin. 2017. From Clarity to Efficiency for Distributed Algorithms. *ACM Transactions on Programming Languages and Systems* 39, 3 (July 2017).

[29] Kenneth L. McMillan. 2016. Modular specification and verification of a cache-coherent interface. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, Ruzica Piskac and Muralidhar Talupur (Eds.). IEEE, 109–116. https://doi.org/10.1109/FMCAD.2016.7886668

[30] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. 252–269.

[31] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon web services uses formal methods. *Commun. ACM* 58, 4 (2015), 66–73.

[32] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Vol. 2283. Springer Science & Business Media.

[33] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. 305–319. https://www.usenix.org/conference/atc14/technical-sessions/

presentation/ongaro

[34] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. 2017. 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

[35] 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, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 614–630.

[36] F. Ramsey. 1930. On a problem in formal logic. In *Proc. London Math. Soc.*

[37] Fred B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)* 22, 4 (1990), 299–319.

[38] ISO TC22/SC3/WG16. 2011. ISO 26262 - Road vehicles — Functional safety. (november 2011).

[39] Klaus v. Gleissenthall, Nikolaj Bjørner, and Andrey Rybalchenko. 2016. Cardinalities and Universal Quantifiers for Verifying Parameterized Systems. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, 599–613.

[40] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. 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, June 15-17, 2015*. 357–368.

[41] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. 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, Saint Petersburg, FL, USA, January 20-22, 2016*, Jeremy Avigad and Adam Chlipala (Eds.). ACM, 154–165. https://doi.org/10.1145/2854065.2854081

A Hoare logic derivation rules

These are the standard rules of Hoare logic that we apply:

$$\frac{T \models (\varphi' \Rightarrow \varphi) \wedge (\psi \Rightarrow \psi') \quad P \vdash_T \{\varphi\}\ \sigma\ \{\psi\}}{P \vdash_T \{\varphi'\}\ \sigma\ \{\psi'\}}$$

$$\frac{P \vdash \{\varphi\}\ \sigma\ \{\psi\} \quad P \vdash \{\psi\}\ \sigma'\ \{\rho\}}{P \vdash \{\varphi\}\ (\sigma; \sigma')\ \{\rho\}}$$

$$\frac{P \vdash \{\varphi \wedge p\}\ \sigma\ \{\varphi\}}{P \vdash \{\varphi\}\ \text{while } p\ \sigma\ \{\varphi \wedge \neg p\}}$$

$$\frac{P \vdash \{\varphi \wedge p\}\ \sigma\ \{\psi\} \quad P \vdash \{\varphi \wedge \neg p\}\ \sigma'\ \{\psi\}}{P \vdash \{\varphi\}\ \text{if } p\ \sigma\ \sigma'\ \{\psi\}}$$

$$\frac{}{P \vdash \{\varphi\langle t/c\rangle\}\ c := t\ \{\varphi\}}$$

$$\frac{}{P \vdash \{\varphi\}\ \text{skip}\ \{\varphi\}}$$

The first is the so-called "rule of consequence". The remainder, respectively, give the semantics of sequential composition, while loops, conditionals, assignments and "skip".