# Modularity for Decidability of Deductive Verification with Applications to Distributed Systems

Marcelo Taube
Tel Aviv University, Israel
mail.marcelo.taube@gmail.com

Giuliano Losa
UCLA, USA
giuliano@cs.ucla.edu

Kenneth L. McMillan
Microsoft Research, USA
kenmcmil@microsoft.com

Oded Padon
Tel Aviv University, Israel
odedp@mail.tau.ac.il

Mooly Sagiv
Tel Aviv University, Israel
msagiv@post.tau.ac.il

Sharon Shoham
Tel Aviv University, Israel
sharon.shoham@gmail.com

James R. Wilcox
University of Washington, USA
jrw12@cs.washington.edu

Doug Woos
University of Washington, USA
dwoos@cs.washington.edu

## Abstract

Proof automation can substantially increase productivity in formal verification of complex systems. However, unpredictablility of automated provers in handling quantified formulas presents a major hurdle to usability of these tools. 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 predictability of proof automation, resulting in a more practical verification approach. We apply this methodology to develop verified implementations of distributed protocols, demonstrating its effectiveness.

***CCS Concepts*** • **Software and its engineering → Formal software verification**;

***Keywords*** Formal verification, Modularity, Decidable logic, Ivy, Distributed systems, Paxos, Raft

**ACM Reference Format:**
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 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3192366.3192414

## 1 Introduction

Verifying complex software systems is a longstanding research goal. Recently there have been some success stories in verifying compilers [29], operating systems [22], and distributed systems [18, 45]. These broadly use two techniques: interactive theorem proving (e.g., Coq [4], Isabelle/HOL [37]) and deductive verification based on automated theorem provers (e.g., Dafny [28] which uses Z3 [11]). However, both techniques are difficult to apply and require a large proof engineering effort. On the one hand, interactive theorem provers allow a user to write proofs in highly expressive formalisms (e.g., higher-order logic or dependent type theory). While this allows great flexibility, it generally requires the user to manually write long and detailed proofs.

On the other hand, deductive verification techniques use automated theorem provers to reduce the size of the manually written proofs. In this approach, user-provided annotations (e.g., invariants, pre- and post-conditions) are used to reduce 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 in practice, e.g., in distributed systems. 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 seemingly irrelevant change in the input causes the prover to fail. As observed in the IronFleet project, SMT solvers can diverge even on tiny examples [18]. 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.

One approach to address the unpredictability of automated solvers is to restrict verification conditions to a decidable fragment of logic [3, 13, 19, 33]. Our previous work on the Ivy verification system [35, 39, 40] has used the effectively propositional (EPR) fragment of first-order logic to verify distributed protocols designs (e.g. cache coherence and consensus). However, the restrictions imposed for decidability are a major limitation. In particular, these restrictions are the reason our previous works only verified protocol designs rather than their executable implementations.

***Decidable Decomposition*** In this paper, we show how to use well-understood modular reasoning techniques to increase the applicability of decidable reasoning and support verifying implementations as well as designs. The key idea is to structure the correctness proof in a modular way, such that each component can be proved using a decidable fragment of first-order logic, possibly with a background theory. Importantly, each component's verification condition can use a *different* decidable fragment. This allows, for example, one component to use arithmetic, while another uses stratified quantifiers and uninterpreted relations. It also allows each component to use its own quantifier stratification, even when the combination would not be stratified. We will refer to this approach as *decidable decomposition*. Crucially, decidable decomposition can be applied even when the global verification condition does not lie in any single decidable fragment (for example, the combination of arithmetic, uninterpreted relations, and quantifiers is undecidable).

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.

***Verifying Distributed Systems*** As a demonstration of decidable decomposition, we verify distributed protocols and their implementations. Distributed protocols play an essential role in today's computing landscape. Reasoning about distributed protocols naturally leads to quantifiers and uninterpreted relations, while their implementations use both arithmetic and concrete representations (e.g., arrays). This combination escapes known decidable fragments. In particular, it prevented our previous work on the Ivy verification system from verifying the implementations of distributed protocols. Here, we overcome this by applying decidable decomposition.

We observe in our evaluation that the human effort needed to achieve decidable decomposition is modest, and follows well known modular design principles. For example, in our implementation of Multi-Paxos, we decompose the proof into an abstract protocol and an implementation, where each component's verification condition falls in a (different) decidable fragment.

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 this paper are:

1. A new methodology, *decidable decomposition*, that uses existing modularity principles to decompose the verification into lemmas proved in (different) decidable logics.
2. A realization of this methodology in a deductive verification tool, Ivy, that supports compilation to C++ and discharges verification conditions using an SMT solver. The fact that all verification conditions are decidable makes the SMT solver's performance more predictable, improving the system's usability and reducing verification effort.
3. An application of the methodology to distributed systems, resulting in verified implementations of two popular distributed algorithms, Raft and Multi-Paxos, obtaining reasonable run-time performance. We show that proofs of these systems naturally decompose into decidable sub-problems. Our experience is that verifying systems in decidable logics is significantly easier than previous approaches.

## 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 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 communicating via asynchronous channels. Even when the code is annotated with invariants, the corresponding verification conditions are expressed in

```
// spec: at most one node
//       sends leader_msg

alreadyvoted := false
voters := ∅
upon client_request() do {
  if ¬alreadyvoted {
    send request_vote_msg(self)
  }
}
upon recv(msg) do {
  if msg.type = request_vote_msg
          ∧¬alreadyvoted {
    alreadyvoted := true;
    send vote_msg(self, msg.src)
  } else if msg.type = vote_msg {
    voters := voters ∪ { msg.src}
    if |voters| > N/2 {send leader_msg(self)}
  }
}
```

**Figure 1.** Toy Leader Election pseudocode.

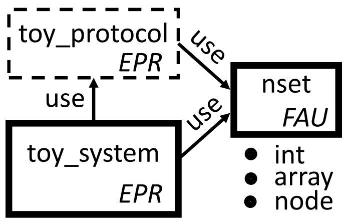

**Figure 2.** Modules and built-in types used for verifying the toy example. Dashed box denotes a ghost module. Each module is annotated with the decidable fragment in which it is verified.

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 [18]. The complexity arises due to the combination of arithmetic, set cardinalities (e.g., number of nodes that voted for a candidate), 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 verification methodology based on decidable reasoning. We use it to develop and verify implementations of distributed systems, whose performance is similar to other verified implementations (e.g. [46]). Rather than starting with an existing implementation (e.g., in C), we define a simple imperative language which permits effective (decidable) reasoning on one hand, and straightforward compilation to efficient C++ code on the other hand.

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 out 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).

## 2.3 Modular Formulation

We illustrate our methodology on the Toy 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 **M**odular **D**ecidable **L**anguage (MDL) — an illustrative programming language enabling modular decidable verification.

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, (2) declarations of other modules and their invariants that are used by the module, specified by the **uses** clause, (3) a module invariant $Q$, given by all **invariant** declarations in the module, (4) procedures with pre-post specifications, specified by **requires** and **ensures** declarations (an unspecified condition is *true* by default), and (5) declaration of the module's initial state, either with **init** declarations or with an init() procedure. Intuitively, a module is correct if all its procedures satisfy their pre/post specification, and also maintain the module invariant, assuming that the used modules are themselves correct. The key property of the modular formulation is that verification conditions generated for each module fall into decidable fragments (see Section 2.4), which allows predictable automation.

We elaborate on the modules of the Toy Leader Election example in Sections 2.3.1 to 2.3.3. Roughly speaking:

(i) The nset module defines (and verifies) a data type which encapsulates sets of nodes under a first-order interface, and hides the low-level implementation. This allows other modules to treat sets of nodes as an opaque type, relying only on the first-order interface, so their verification can be carried out in uninterpreted first-order logic. Verifying that the nset module satisfies its interface is carried out in a suitable theory. The first-order interface includes a predicate that tests if a set of nodes forms a majority, along with the property that any two majorities intersect, which is crucial for the proof of the protocol.

(ii) The toy_protocol module defines (and verifies) an abstract model of the protocol, eliding some of the implementation details. This is in line with the common practice of developing a system in an evolutionary process: starting with a design, and then gradually developing an efficient implementation. Here this practice serves a different purpose; namely, the toy_protocol captures the protocol 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 decompose the verification problem, and as we shall see, to avoid quantifier alternation cycles.

```
1  ghost module toy_protocol uses nset, nset.majorities_intersect {
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  procedure vote(v : node, n : node) {
12   requires ∀n′.¬voted(v, n′)
13   voted(v, n) := true
14  }
15  procedure become_leader(n : node, s : nset.t) {
16   requires nset.majority(s) ∧ ∀n′ : node. nset.member(n′, s) → voted(n′, n)
17   isleader(n) := true
18   quorum := s
19  }
20 }
```

**Figure 3.** Protocol module for Toy Leader Election.

(iii) The toy_system module specifies (and verifies) the system implementation, using both the data type defined by nset (relying on its first-order specification) and the abstract protocol defined by toy_protocol (as ghost code) to obtain a verified executable 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. When proving this module, the majority intersection invariant of the nset module is used, as indicated by the **uses** clause in line 1.

The module also provides two procedures that define the abstract protocol steps. Each procedure specifies a pre condition. 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 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.

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

**Figure 4.** System module for Toy Leader Election.

### 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. The invariant on line 6 specifies the desired specification, i.e., that only one node ever issues a leader_msg message. This is the ultimate guarantee provided by the implementation, and it is the only line of trusted specification in the example. Namely, one has to trust that this invariant captures the intended property (e.g., by careful inspection), but the fact that the implementation maintains it is mechanically verified. The declarations are followed by the code to be run on each node. This code (lines 8 to 30) defines the local state of each node, as well as procedures that can be executed in

```
 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.** The nset module for node sets, proving the majority intersection property.

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.

We note that the toy_system module makes use of the nset module to maintain sets of voters, and uses a majority test (line 26). It also makes calls into the ghost module toy_protocol (lines 21 and 28). This allows to establish an invariant that relates the state of the concrete module and the abstract module (lines 32 to 35), and use the proven invariant of the abstract module as a lemma for proving the concrete implementation. This is indicated by the **uses** clause, which declares use of the toy_protocol.one_leader invariant (line 1).

### 2.3.3 Node Set 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 11). 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 (the $\lambda$ at line 27 is eliminated from verification conditions by $\beta$-reduction, resulting in first-order formulas; see Section 4.6). 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 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

We now explain how verification conditions are generated for each module, and how they are checked under (possibly different) theories. We use two decidable fragments: the effectively propositional (EPR) fragment [41] which allows stratified quantifier alternations, and the finite almost uninterpreted (FAU) fragment [17] which includes linear integer arithmetic in a restricted way.

***Verification Condition Generation*** Recall that each module declares the modules and invariants it uses in the **uses** clause. Based on this, verification conditions are derived automatically. Each module must provide the following *guarantees*: 1) The module invariant, $Q$, holds at any initial state. 2) Every procedure in the module establishes its postcondition and preserves the module invariant $Q$, and 3) At each call site, the precondition of the called procedure is established. Each module may rely upon the following *assumptions*: 1) Every called procedure establishes its postcondition and 2) Every used invariant of another module holds at all times. The verification condition states that the assumptions imply the guarantees.

As an example, the verification condition generated for the become_leader procedure (Figure 3 line 15) is:

$$\Big(Q_{\text{toy\_protocol}} \wedge Q_{\text{nset.majorities\_intersect}} \wedge$$
$$(\text{majority}(s) \wedge \forall n'.\, \text{member}(n',s) \rightarrow \text{voted}(n',n))\Big) \rightarrow$$
$$Q_{\text{toy\_protocol}}\,[(\text{isleader}(x) \vee x = n)\,/\,\text{isleader}(x)\,,\, s\,/\,\text{quorum}]$$

where $Q_{\text{toy\_protocol}}$ is given by Figure 3 lines 7 to 9 and $Q_{\text{nset.majorities\_intersect}}$ is given by Figure 5 line 11. The latter is used because it appears in the **uses** clause of the toy_protocol module. The verification condition also includes the procedure precondition taken from Figure 3 line 16, and checks that assuming the invariants and the precondition, the invariant $Q_{\text{toy\_protocol}}$ is preserved by the procedure (the substitutions reflect the assignments of lines 17 and 18).

Optionally, calls to procedures may be inlined (i.e., replaced by the procedure body), and we may take the initial condition of another module as an assumption. If we use a module in this way, we say the module is *opened*. As an example, ghost module toy_protocol is opened in the verification of toy_system (Figure 4, line 36). This allows us to establish easily an invariant relating the states of the two modules. When composing modules, there are additional conditions required for soundness (e.g., that modules do not interfere), which are described in Section 4.

***Theory Abstractions*** Recall that every module may include interpreted types (e.g., int, array) as well as definitions via **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, which may be viewed as a form of abstraction.

For example, when checking the verification condition of the become_leader procedure given above, the majority and member relations are treated as uninterpreted relations, and not according to their definitions from Figure 5 lines 8 and 9. In contrast, when verifying the nset module, these definitions are used as part of a background theory, which also includes linear integer arithmetic.

***Decidable Decomposition of Toy Leader Election*** For the whole picture of our example, observe that 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 combine arithmetic, quantifier alternation cycles, and uninterpreted symbols, 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 that 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.

In terms of theories, the nset module is verified when int is interpreted using the theory of linear integer arithmetic, and the nodeset.t type is interpreted as array<int,node> from our built-in array theory (as explained in Section 4, we encode arrays in FAU). Moreover, the member and majority relations are interpreted by their definitions (Figure 5 lines 8 and 9). The resulting verification conditions for this module are in FAU. When verifying the toy_protocol and toy_system modules, sorts and relations (including member and majority) are uninterpreted. The resulting verification conditions are in EPR, as the quantifiers in each module are stratified.

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 pure function closures equipped with a memo table. Every type in a non-ghost module must have one of the above as its interpretation.

Arrays in Ivy are immutable, so procedures manipulating them (e.g., append) return a new array object. For efficient execution, the compiler optimizes cases where the modification

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

can be implemented in place, which is common in our examples. In particular, all array manipulations in the Toy Leader Election example are compiled to in-place modifications.

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.

***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 operator is trusted to run the system with a correct configuration, 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). Ultimately, the trusted base of the verified system includes the Ivy verifier and compiler (including Z3), the implementation of built-in types and the shim, and the operator's configuration process.

## 3 Preliminaries

### 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 $\Sigma$ 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 $\Sigma$-structure maps each sort $t \in \mathcal{S}$ to a non-empty set called the *universe* of $t$, and each symbol $s : t$ in $\Sigma$ 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 $\Sigma$ 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 $\Sigma$. 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 $\sigma$ can be extended to a $\Sigma$-structure $\sigma'$ such that $\sigma' \models T$.

Intuitively speaking, a $\mathcal{V}$-conservative theory $T$ can be used to extend the vocabulary with symbols in $\mathcal{V}$, possibly defining them in terms of other symbols. 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 $\Sigma$ that contains only constant and relation symbols, and where all sorts and symbols are uninterpreted. Satisfiability of EPR formulas is decidable [30]. 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 *quantifier alternation 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 $t$ as its range. Notice that a formula of the form $\forall x : s.\, \exists y : t.\, \varphi$ has an edge $(s, t)$ in its quantifier alternation graph, since the Skolem function for $y$ is of sort $s \to t$. A *bad cycle* in the quantifier alternation graph of $\varphi$ is one containing a sort $s$ such that some variable of sort $s$ is universally quantified in the Skolem normal form of $\varphi$.

A formula is *stratified* if its quantifier alternation 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 $\varphi$ is *virtually stratified* if there is *any* consistent assignment of sorts to symbols in $\Sigma(\varphi)$ under which $\varphi$ is stratified. As an example, the formula $\forall x : s.\ \exists y : t.\ f(x) = y$ is stratified, since the quantifier alternation 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 [17]).

***Finite Almost Uninterpreted Fragment (FAU)*** Formulas in the almost uninterpreted fragment [17] 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 $\forall x : \mathrm{int}.\ x + y \leq z$, is not in the fragment, since the variable $x$ appears under the interpreted relation $\leq$. However $\forall x : \mathrm{int}.\ f(x) + y \leq z$ is allowed, as is $\forall x : \mathrm{int}.\ x \leq y$.

The *finite* almost interpreted fragment (FAU) is defined as the set of almost interpreted formulas that are stratified as defined in [17]. Satisfiability of FAU modulo the theory is decidable. In particular, in [17] 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 [6].

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.

## 4 Modular Proofs

In this section we describe an illustrative modular reasoning system, using a very simple procedural language as a model of MDL. This system is not as rich as the system actually used in the Ivy tool but is sufficient to capture the proof strategies we apply here.

### 4.1 A Model Language

Let $\Sigma$ be a vocabulary of non-logical symbols. The set of propositions (terms of Boolean sort) over $\Sigma$ is denoted $\mathcal{P}$.

#### 4.1.1 Statements

The statements in our model language are defined as follows:

**Definition 4.1.** Let $\mathcal{N}$ be a set of *procedure names* and $\mathcal{V}_P \subseteq \Sigma$ 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 $\Sigma$, $\tau$ a sort, $p \in \mathcal{P}$, and $n \in \mathcal{N}$.

The mutable program variables $\mathcal{V}_P$ are a subset of $\Sigma$. The statements have the expected semantics. For now, program variables $c$ are restricted to logical constants, and can only be assigned terms $t$ of first-order types. We will relax this restriction in Section 4.6.

#### 4.1.2 Procedures and Modules

The *Hoare triples* $\mathcal{H}$ are denoted $\{\varphi\}\ \sigma\ \{\psi\}$, where $\varphi, \psi \in \mathcal{P}$ and $\sigma \in \mathcal{S}$. 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 $\varphi$ of a procedure is introduced by the **requires** keyword and the postcondition by **ensures**.

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 $\mathrm{pre}(f)$ for its pre-image and $\mathrm{img}(f)$ for its image. If $P$

is a context, we will write called($P$) for the set of names $n$ such that "call $n$" occurs in $P$.

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

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

That is, $P$ gives a set of procedure definitions with pre/post specifications, $E$ gives the subset of these definitions that is exported to the environment, $I$ is a set of predicates that are true in the module's initial state, and $\wedge Q$ is an inductive invariant that holds between calls to exported procedures. In the sequel, we often use $I$ to denote $\wedge I$ and $Q$ to denote $\wedge Q$. We will write $P_M$, $E_M$, $I_M$, $Q_M$, respectively, for the components of module $M$. In MDL, a **module** declaration creates a module, with all procedures exported by default. The initial condition $I_M$ is specified by **init** declarations. (In Section 4.6, we allow a module to define an init() procedure instead of an initial condition.) The invariant $Q_M$ is given by the set of **invariant** declarations in the module.

We define the following operations on modules:

**Definition 4.4.** Let $M = (P,E,I,Q)$ and $M' = (P',E',I',Q')$ be modules such that $\text{pre}(P) \cap \text{pre}(P') = \emptyset$ and $\Sigma(I) \cap \Sigma(I') \cap \mathcal{V}_P = \emptyset$. The *composition* of $M'$ and $M$, denoted $M' + M$, is $(P \cup P', E \cup E', I \cup I', Q \cup Q')$.

**Definition 4.5.** For a module $M = (P,E,I,Q)$ and a set $E' \subseteq \mathcal{N}$, the *restriction* of $M$ to $E'$, denoted $M \downarrow E'$, is the module $(P, E \cap E', I, Q)$.

## 4.2 Axiomatic Semantics

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

The axiomatic semantics of the statements of our language is given by the standard rules of Hoare logics:

$$\textsc{Cons}\ \frac{\begin{array}{c} T \models (\varphi' \Rightarrow \varphi) \wedge (\psi \Rightarrow \psi') \\ P \vdash_T \{\varphi\}\ \sigma\ \{\psi\} \end{array}}{P \vdash_T \{\varphi'\}\ \sigma\ \{\psi'\}}$$

$$\textsc{Comp}\ \frac{\begin{array}{c} P \vdash \{\varphi\}\ \sigma\ \{\psi\} \\ P \vdash \{\psi\}\ \sigma'\ \{\rho\} \end{array}}{P \vdash \{\varphi\}\ (\sigma;\sigma')\ \{\rho\}}$$

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

$$\textsc{If}\ \frac{\begin{array}{c} P \vdash \{\varphi \wedge p\}\ \sigma\ \{\psi\} \\ P \vdash \{\varphi \wedge \neg p\}\ \sigma'\ \{\psi\} \end{array}}{P \vdash \{\varphi\}\ \text{if}\ p\ \sigma\ \sigma'\ \{\psi\}}$$

$$\textsc{Assign}\ \frac{}{P \vdash \{\varphi[t\ /\ c]\}\ c := t\ \{\varphi\}}$$

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

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

The first is the so-called "rule of consequence". The remainder, respectively, give the semantics of sequential composition, while loops, conditionals, assignments and "skip". The last rule is the Inline rule that gives the semantics of non-recursive procedure calls: 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 $\varphi'$ and may use the specified post-condition $\psi'$. 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 we treat here.

We will write $I; P \vdash M$ to represent the judgment that, in context $P$, if formulas $I$ hold initially, then module $M$ maintains its invariant and satisfies its pre/post specifications. It is assumed that the environment only calls $M$'s exported procedures, and otherwise never modifies its program variables. We elide $I$ or $P$ if they are empty sets. The axiomatic semantics of modules is given by the following Module rule:

$$\textsc{Module}\ \frac{\begin{array}{ll} \multicolumn{2}{c}{I, I_M \models Q_M} \\ \multicolumn{2}{l}{\text{for } n{:=}\{\varphi\}\ \sigma\ \{\psi\} \text{ in } P_M\text{:}} \\ \quad P, P_M \vdash \{\varphi\}\ \sigma\ \{\psi\} & \text{if } n \in \text{called}(P_M) \\ \quad P, P_M \vdash \{Q_M \wedge \varphi\}\ \sigma\ \{Q_M \wedge \psi\} & \text{if } n \in E_M \end{array}}{I; P \vdash M}$$

In this rule, $Q_M$ 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. In addition, internally called procedures must satisfy their specifications *without* assuming the invariant, since the invariant may be violated during execution of the module's procedures.

## 4.3 Rules for Decidable Decomposition

The rules defined in Section 4.2 provide the full axiomatic semantics. In particular, they enable to verify a program which consists of multiple modules, say $M_1, \ldots, M_n$, and exports procedures $E$ to its environment, by proving $\vdash (M_1 + \ldots + M_n) \downarrow E$. However, the class of verification conditions generated is undecidable. In this section, we provide *derived* inference rules that can be used to decompose the proof such that the verification conditions are in a decidable fragment. For simplicity, we only present the rules needed for the Toy Leader Election example. Our implementation includes more flexible rules that are similar in spirit.

First, we introduce a rule that allows to verify a procedure call without inlining the procedure's body, by using the assumption that the procedure satisfies its pre/post specification at the call site:

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

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

The next rules allow to verify the composition of modules by verifying the individual modules. We begin by defining some notation.

**Definition 4.7.** The *callset* $\text{calls}(M,P)$ of module $M$ in context $P$ is the least set of procedures $n$ such that either $n$ is exported from $M$, or $n$ is in $\text{pre}(P_M \cup P)$ and $n$ is called from $\text{calls}(M,P)$. Formally, $\text{calls}(M,P) = \mathit{LFP}.\,\lambda X.\,E_M \cup X \cup (\mathit{called}(X) \cap \text{pre}(P_M \cup P))$. The *refset* $\text{ref}(M,P)$ is the subset of $\mathcal{V}_P$ occurring in $\text{calls}(M,P)$ or in $I_M$. The *modset* $\text{mod}(M,P)$ is the subset of $\mathcal{V}_P$ *assigned* in $\text{calls}(M,P)$, or occurring in $I_M$. Module $M$ is said to *interfere* with module $M'$ in context $P$, denoted $M,P \rightsquigarrow M'$, if $\text{mod}(M,P) \cap \Sigma(Q_{M'}) \neq \emptyset$ or if $\text{called}(\text{calls}(M,P)) \cap \text{pre}(P_{M'}) \not\subseteq E_{M'}$.

In other words, $M$ interferes with $M'$ if it either modifies a variable occurring in the invariant of $M'$, or if it calls an internal procedure of $M'$. Module $M$ can interfere with $M'$ directly, or by calling procedures defined in the context $P$.

To export an invariant from one module to another, we introduce two notations. If $M$ is a module and $\Gamma$ is a set of formulas, we say $M[\Gamma]$ is the module that results from conjoining $\wedge\Gamma$ to the postcondition of every exported procedure of $M$. On the other hand, $[\Gamma]M$ results from conjoining $\wedge\Gamma$ to the precondition of every exported procedure of $M$.

Now we can derive a compositional rule that allows us to verify a service $M'$ layered on a service $M$, while *assuming* the invariants and pre/post specifications of $M$.

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

$$\textsc{Layer}\,\frac{\begin{array}{c} I;P \vdash M \\ I,\Theta;\ P,P_{M[\Gamma]} \vdash [\Gamma]M' \end{array}}{I;P \vdash (M+M') \downarrow E}\ \begin{array}{l} M' \downarrow E,P \not\rightsquigarrow M \\ M \downarrow E,P \not\rightsquigarrow M' \\ \Gamma \subseteq Q_M \\ \Theta \subseteq I_M \end{array}$$

Ignore for the moment the expressions in square brackets. The rule states that, to verify $M'$ layered on $M$, we first verify $M$, then verify $M'$ assuming the proved specifications of the exported procedures of $M$. Intuitively, this works because external calls to one module cannot invalidate the invariant of the other. Note the asymmetry, however. Module $M$ must be proved in context $P$, which means that it contains no calls to procedures outside of $P$, and in particular, no call-backs into $M'$. At a call-back, the invariant of $M'$ would not hold, violating the precondition under which $M'$ is verified. Technically, this rule can be derived by annotating every statement of $M'$ with the invariants of $M$. The rule also allows us to use initial conditions of $M$.

The bracketed expressions allow us to assume the proved invariants of $M$ when proving $M'$. We do this by assuming these invariants on entry to every exported procedure of $M'$ and on exit of every exported procedure of $M$.

## 4.4 Ghost Modules and Slicing

**Definition 4.9.** If $P$ is a context, the *slice* of $P$, denoted $\text{slice}(P)$ is the set of procedure definitions which contains $n{:=}\{\mathit{true}\}\text{skip}\{\mathit{true}\}$ for each $n{:=}\{\varphi\}\sigma\{\psi\}$ in $P$. If $M$ is a module, $\text{slice}(M)$ denotes $(\text{slice}(P_M), E_M, \emptyset, \emptyset)$.

The following derived rule can be used to slice out a "ghost" module that is used only for the purpose of the proof. We say a module $M$ is *invisible* to $M'$ in context $P$, denoted $M,P \not\leftrightarrow M'$, if $M,P \not\rightsquigarrow M'$ and $\text{mod}(M,P) \cap \text{ref}(M',P) = \emptyset$ and $I_M$ is $\mathcal{V}_P$-conservative (definition 3.1) and every exported procedure of $M'$ terminates in context $P$, starting in all states satisfying its precondition.

**Theorem 4.10.** *The following rule can be derived:*

$$\textsc{Slice}\,\frac{P \vdash (M+M') \downarrow E_{M'}}{P \vdash (\text{slice}(M)+M') \downarrow E_{M'}}\,M,P \not\leftrightarrow M'$$

To prove termination for the examples presented here, it suffices to verify that there is no recursion and that $P_M$ contains no "while" statements (Ivy supports proof of termination using a ranking). We must also show that every model of the theory has 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 invisible module $M$ can be used like a lemma in the proof of $M'$. That is, we make use of its properties and then discard it, as we did with the abstract protocol model in Toy Leader Election.

## 4.5 Theory Abstractions

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

**Theorem 4.11.** *The following rule can be derived:*

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

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 nset.majority when verifying the abstract model and the implementation, but used them when verifying the nset module.

## 4.6 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 [40]. 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 and bit vectors (both with the usual arithmetic operators). In Ivy, these theories are provided natively by Z3. Finite immutable arrays are provided as an abstract data type, with functions provided for length and select, and axiomatically specified procedures for update and element append. For each finite sort $\tau$ (such as node in Toy Leader Election) the language provides an array constant $\tau$.all that contains all elements of sort $\tau$. We used this feature to define the notion of a majority of nodes in Toy Leader Election.

A module $M$ may have a special initialization procedure $M$.init() that is called by the environment before any exports. In this case, the Module rule is modified to require that this procedure establishes the module invariant with no precondition (as it does, for example, in the nset module).

## 4.7 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 $\mu$ of mesages transmitted on the network, we introduce an abstract relation sent$(m : \mu)$ to represent the fact that message $m$ has been broadcast in the past. We add to the language a primitive "send $m : \mu$" whose semantics is defined by the following rule:

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

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

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

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.

## 4.8 Proof of Toy Leader Election

To illustrate the inference rules, we explain how to prove Toy Leader Election by chaining them. Let $M^1, M^2, M^3$ be, respectively, the modules nset, toy_protocol and toy_system. First, we prove $\vdash_T M^1$ where theory $T$ consists of the integer arithmetic and array theories, and the definitions of the majority and member relations. Next we prove $P_{M^1[\Gamma]} \vdash [\Gamma]M^2$ (using the Module and Call rules). Here, $\Gamma$ is the majority intersection invariant of nset. Notice we do not use theory $T$, to preserve decidability. We then add theory $T$ using the Theory rule, and combine with the above using the Layer rule, to obtain $\vdash_T M^1 + M^2$. Then we prove:

$$I_{M^2}; P_{(M^1+M^2)[\Gamma']} \vdash [\Gamma']M^3$$

Here $\Gamma'$ is the invariant of toy_protocol used by toy_system. By separating the proof of this invariant, we avoided a function cycle. In this proof, we inline the procedures of the abstract model $M_2$ and use the Send rule to capture the semantics of message sending. Again using the Theory and Layer rules, we obtain $\vdash_T (M_1 + M_2 + M_3) \downarrow E_{M^3}$. We use the Slice rule to remove the ghost module, obtaining $\vdash_T (M_1 + \text{slice}(M_2) + M_3) \downarrow E_{M^3}$. Finally, the network rule hides the message handlers, giving the conclusion:

$$\vdash_T^{\mathcal{M}} (M^1 + \text{slice}(M^2) + M^3) \downarrow \{\text{request\_vote}\}\,.$$

This leaves request_vote as the only exported procedure, which is called in responce to a client request to initiate the protocol. The result is a verified equivalent to the code of Figure 1. Note that these steps are not written explicity, and are inferred from **uses**, **open**, **ghost**, and **system** directives.

## 4.9 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 and the Module rule. In reality, 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 [31] (similarly to, e.g., IronFleet [18]). 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.

### 4.10 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 [28]. 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 quantifer alternation 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 [38] and Multi-Paxos [26][2]. Both protocols implement a centralized shared log abstraction, i.e., a write-once map from indices to values, which can be used to implement a fault-tolerant distributed service using state machine replication (SMR) [42]. We verify that no two replicas ever disagree on the committed portion of the replicated log. We implemented only the basic Raft and Multi-Paxos protocols, without log truncation, state transfer to slow replicas, persisting the log to disk, crash recovery, batching, or application-level flow control.

### 5.1 Verifying 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. Our decidable decomposition consists of a main module that contains the core protocol logic, and of a module that separates the local node state from the main module and exposes only relations in order to avoid quantifier alternations. To separate theories, Raft also uses the nset module presented in Section 2 (for node sets with majority testing), and a module that implements a log data type, internally represented using Ivy's built-in arrays. This strategy uses the modular reasoning principles explained in Section 4 in a different way than the strategy used in Section 2 for the Toy Leader Election example: instead of separating the abstract protocol from the implementation, it separates the implementation from the representation of the local state.

The main module contains the message handlers, which implement Raft's logic, as well as an inductive invariant that proves safety. This invariant is proved using the majority intersection property, which introduces a quantifier alternation edge from node set to node. The state of each node is naturally represented by a function from node to various sorts (including node set, e.g., to let each node track its voters). Combining such functions in the main module would create quantifier alternation cycles (e.g., between node and node set). To avoid the cycles, we encapsulate the local state of nodes in a separate module that exposes only relations. For example, a node's current term is concretely represented by a function $t$ : node → term, and it is exposed as a relation $r$ : node, term, intended to capture $r(n,x) \equiv (t(n) = x)$, and an additional relation that captures $t(n) \geq x$.

Verification of both the main module and the local state module is done in EPR, with log indices and terms abstracted as linear orders. Crucially, although both modules use EPR, they use different quantifier alternation stratification orders, so a non-modular proof would fall outside the decidable fragment. The modules for node sets and logs are verified in the FAU fragment, which allows the necessary reasoning about arithmetic.

***Multi-Paxos*** Our approach to implementing and verifying Multi-Paxos follows the strategy presented in Section 2 for the Toy Leader Election example. We separate the abstract protocol and its proof into a ghost module, and use this module as a lemma for proving the system implementation.

Similarly to Raft, our Multi-Paxos implementation uses the nset module, which provides an abstract data type of node sets with majority testing. However, the majority intersection property (Figure 5 line 11) is used only in the proof of the abstract protocol, and is not used when verifying the implementation module. Therefore, there is no quantifier alternation edge from quorum to node in the VC's of the implementation module, and, contrary to Raft, there is no need to abstract the local state of the nodes using relations.

### 5.2 Verification Effort

The Raft verification took approximately 3 person-months. The code and proof of sum up to 840 lines, among which 300 consist of invariants and ghost code. This gives a proof-to-code ratio of 0.6 for Raft. Obtaining the Multi-Paxos implementation from the abstract protocol (which was already proved in [39]) took approximately two person-months of work. The Multi-Paxos code and proof sum up to 789 lines,

[2]The artifact is available at https://www.cs.tau.ac.il/~marcelotaube/modularity-for-decidability.html

166 of which consist of invariants and ghost code. This gives a proof-to-code ratio of 0.3 for Multi-Paxos. Ivy successfully discharges all VC's of both Raft and Multi-Paxos in 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 [18] implementation is part of a verification effort of 3.7 person-years (which also includes IronKV, a verified key-value store). Excluding generic libraries such as verified serialization code and liveness proofs, which are not verified in the Ivy implementations, IronRSL consists of roughly 3,000 implementation and 12,000 proof SLOC. This gives a proof-to-code ratio of 4. IronFleet's VC checking was performed on cloud servers to obtain verification times acceptable for interactive use.

As evidenced by its larger code-base, IronRSL includes more features than the Ivy implementations presented: log truncation, batching, and state-transfer are part of IronRSL (however IronRSL does not support crash recovery). Moreover, more properties are verified compared to the Ivy implementations: IronFleet verifies the network serialization and deserialization code, some liveness properties, and the model includes resource bounds (e.g., integer overflows).

The Verdi proof of the Raft protocol [46] consists of 50,000 proof and 530 implementation SLOC, and required several person-years. Among those 50,000 lines, most are devoted to manual proofs of lemmas directly required for the Raft verification, and are thus not immediately generalizable to other protocols. Generic pieces of code, such as the network semantics, make up less than 5% of the code base.

## 5.3 Verified System's Performance

Our implementations of Raft and Multi-Paxos are compiled by Ivy to C++, and the resulting code is linked with several trusted libraries, including a low-level networking interface based on TCP using non-blocking sends. In order to evaluate the performance of these verified replication protocols, we implemented a key-value store that can use either protocol internally. Unverified components handle the key-value store state machine and client communication on top of the verified replication library. The unverified portion consists of 785 SLOC of C++. We benchmarked the performance of these systems against that of two other Raft-based key-value stores: `vard` [45], a similar verified system, and `etcd` [10], an unverified, production-quality system. Our goal in these experiments is to show that systems developed with our decidable verification methodology achieve comparable performance to existing systems.

We benchmarked all systems on a cluster of three Amazon EC2 `m4.xlarge` nodes, each with 4 (virtual) CPU cores and 16GB of RAM. The cluster served 16 closed-loop client processes running on a fourth `m4.xlarge` node. The client processes sent a randomized 50/50 mix of `GET` and `PUT` requests to the servers, and we measured the observed throughput and latency. The results are summarized in Figure 6. We chose 16 client processes through additional experimentation that revealed increasing the load further does not increase throughput, and instead causes the systems to become overloaded, resulting in performance degradation and instability. Since our systems do not include application-level flow control, they do not gracefully handle overload conditions.

| System | Throughput [req/ms] | Latency [ms] |
|---|---|---|
| Ivy-Raft | 13.5 | 1.2 |
| Ivy-Multi-Paxos | 8.7 | 1.9 |
| `vard` | 4.4 | 3.7 |
| `etcd` | 9.2 | 1.7 |

**Figure 6.** Performance of SMR-based key-value stores.

While the performance of all systems is roughly in the same order of magnitude, there are some expected differences. The systems vary in the languages, libraries, and data structures used, and they implement different optimizations. The Multi-Paxos implementation notifies replicas of decision by broadcasting "decide" messages. In Raft, those messages are instead piggy-backed on subsequent requests, resulting in more efficient network usage. Unlike our implementations, `vard` and `etcd` were designed to persist data to disk. We partially mitigated this difference by modifying `vard` to disable disk writes and configuring `etcd` to use a RAM disk. Our conclusion from this experiment is that systems verified with decidable decomposition can achieve similar performance to existing systems (we do not claim that any of the systems considered outperforms another).

The results above were obtained while the systems were operating normally, i.e., a single elected leader and no failures. To test their fault tolerance, we also measured our systems during leader failure and reelection. After startup, we let the system elect a leader and begin servicing client requests, until a steady state is reached. We then killed the server process on the leader node and observed the time required before clients could be serviced again. In both Raft and Multi-Paxos, the system recovered in 4-5 seconds. This delay is expected due to the way timeouts are currently set in our implementation, and could be improved by additional engineering.

## 6 Related Work

***Fully Automatic Verification*** Fully automatic verification of distributed protocols and systems is usually beyond reach because of undecidability. Bounded checking is successfully used, e.g., in Alloy [21] and TLA+ [27], to check correctness of designs up to certain numbers of nodes. This is useful, due to the observation that most bugs occur with small numbers of nodes. However as observed by Amazon [36] 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. Another approach for automation is to use sound and incomplete procedures for deduction and invariant search for logics that combine quantifiers and set cardinalities [14, 44]. However, distributed systems of the level of complexity we consider here (i.e., Raft, Multi-Paxos) are beyond the reach of these techniques. 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 usually left without any solution.

***Interactive Verification*** The IronFleet [18] and Verdi [45] projects recently demonstrated that distributed systems can be proved all the way from design to implementation. The DistAlgo project [7, 32] develops programming methodologies for interactively verifying distributed systems using the TLA+ proof system [8]. However, interactive verification requires tremendous human efforts and skills. 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 tools, since one does not need to understand how decision procedures such as SMT work. While this work considers invariant in first-order logic, it may be possible aplly its principles to richer 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. The Threshold Automata formalism and the ByMC verification tool [5, 23–25] allow to express a restricted class of distributed algorithms operating in a partially synchronous communication mode. This restriction allows decidability results based on cutoff theorems, for both safety and liveness. Recently, [34] presented a cutoff result for consensus algorithms. This allows to verify, e.g., the core Paxos algorithm, using a cutoff bound of 5 processes. However, this work is focused on algorithms for the core consensus problem, and does not support infinite-state per process, that is needed, e.g., to model Multi-Paxos and SMR. 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.

***Modularity in Verification*** The utility of modularity for simplified reasoning was already recognized in the seminal works of Hoare and Dijkstra (e.g., [20]). Proof assistants such as Isabelle/HOL [37] and Coq [4] provide various modularity mechanisms. Deductive verification engines such as Dafny [28] and VCC [9] employ modularity to simplify reasoning in a way that is similar to ours. The program logic Disel [43] allows modular verification of distributed systems, with rules somewhat similar to ours. In [16], a series of transformations are applied to obtain a verified implementation of Multi-Paxos from single-decree Paxos in a compositional manner. 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 our methodology is useful for verifying complex distributed 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. Such implementations involve arithmetic, unbounded sets of processes, 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.

***Acknowledgements*** We thank Aurojit Panda, our shepherd Adam Chlipala, the anonymous referees, and the anonymous artifact evaluation referees for their insightful comments. Padon was supported by a Google PhD fellowship. This publication is part of projects that have received funding from the European Research Council (ERC) under the European Union's Seventh Framework Program (FP7/2007–2013) / ERC grant agreement no. [321174-VSSC], and Horizon 2020 research and innovation programme (grant agreement No [759102-SVIS]). The research was partially supported by Len Blavatnik and the Blavatnik Family foundation, the Blavatnik Interdisciplinary Cyber Research Center, Tel Aviv University, and the Pazi Fund. This material is based upon work supported by the National Science Foundation under Grant No. 1655166, and by the United States-Israel Binational Science Foundation (BSF) grants No. 2016260 and 2012259.

## 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] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2004. A Decidable Fragment of Separation Logic. In *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference, Chennai, India, December 16-18, 2004, Proceedings.* 97–109.

[4] 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

[5] 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

[6] 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

[7] 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.

[8] 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.

[9] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.), Vol. 5674. Springer, 23–42. https://doi.org/10.1007/978-3-642-03359-9_2

[10] CoreOS 2014. etcd: A highly-available key value store for shared configuration and service discovery. 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] Bruno Dutertre, Dejan Jovanović, and Jorge A. Navas. 2018. Verification of Fault-Tolerant Protocols with Sally. In *NASA Formal Methods*, Aaron Dutle, César Muñoz, and Anthony Narkawicz (Eds.). Springer International Publishing, Cham, 113–120.

[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] Álvaro García-Pérez, Alexey Gotsman, Yuri Meshman, and Ilya Sergey. 2018. Paxos Consensus, Deconstructed and Abstracted. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings.*

[17] 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.

[18] 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.

[19] Jesper G. Henriksen, Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. 1995. Mona: Monadic Second-Order Logic in Practice. In *Tools and Algorithms for Construction and Analysis of Systems, First International Workshop, TACAS.* 89–110.

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

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

[22] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2010. seL4: formal verification of an operating-system kernel. *Commun. ACM* 53, 6 (2010), 107–115.

[23] 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.

[24] 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.

[25] 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

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

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

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

[29] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

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

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

[32] 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).

[33] P. Madhusudan, Gennaro Parlato, and Xiaokang Qiu. 2011. Decidable logics combining heap structures and data. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. 611–622.

[34] Ognjen Maric, Christoph Sprenger, and David A. Basin. 2017. Cutoff Bounds for Consensus Algorithms. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II (Lecture Notes in Computer Science)*, Rupak Majumdar and Viktor Kuncak (Eds.), Vol. 10427. Springer, 217–237. https://doi.org/10.1007/978-3-319-63390-9_12

[35] 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

[36] 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.

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

[38] 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

[39] 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

[40] 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.

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

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

[43] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and proving with distributed protocols. *PACMPL* 2, POPL (2018), 28:1–28:30.

[44] 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.

[45] 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.

[46] 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