High-Level Executable Specifications of Distributed Algorithms

Yanhong A. Liu, Scott D. Stoller, and Bo Lin

Computer Science Department, State University of New York at Stony Brook
{liu,stoller,bolin}@cs.stonybrook.edu

Abstract. This paper describes a method for specifying complex distributed algorithms at a very high yet executable level, focusing in particular on general principles for making properties and invariants explicit while keeping the control flow clear. This is critical for understanding the algorithms and proving their correctness. It is also critical for generating efficient implementations using invariant-preserving transformations, ensuring the correctness of the optimizations.

We have studied and experimented with a variety of important distributed algorithms, including well-known difficult variants of Paxos, by specifying them in a very high-level language with an operational semantics. In the specifications that resulted from following our method, critical properties and invariants are explicit, making the algorithms easier to understand and verify. Indeed, this helped us discover improvements to some of the algorithms, for correctness and for optimizations.

1 Introduction

Distributed algorithms are at the core of distributed systems, which are increasingly indispensable in our daily lives. Yet, understanding and proving the correctness of distributed algorithms remain challenging, recurring tasks. Study of distributed algorithms has relied on either pseudo code with English, which is high-level but imprecise, or formal specification languages, which are precise but harder to understand or not executable.

For example, the well-known Paxos algorithm for distributed consensus, from when Lamport first described it in 1990 [16], through all the variations, investigations, and practical deployments (including Google's Chubby distributed locking and storage service [6]) over the years, e.g., [8, 17, 5], remains as actively studied as ever in specification and verification, e.g., [20, 33]. The description by van Renesse [33] finally provides precise pseudo code for full Paxos—multi-Paxos—with comprehensive detailed explanations.

This paper describes a method to help make it easier to understand and verify complex distributed algorithms by specifying them at a very high yet executable level. The method focuses in particular on general principles for making properties and invariants explicit while keeping the control flow clear. It exploits message history sequences and queries over sets and sequences to abstract the handling of received messages, and to abstract synchronization, when to send what messages to whom, and sending of messages collectively.

Making properties and invariants explicit is critical also for generating efficient implementations using invariant-preserving transformations, ensuring the correctness of the optimizations. In fact, it was during the study of these optimizations in the last several years, while trying to better understand and teach distributed algorithms, that we developed the abstractions and the specification method.

We have studied and experimented with a variety of important distributed algorithms, including well-known difficult variants of Paxos, by specifying them in a very high-level language with an operational semantics. In the specifications that resulted from following our method, critical properties and invariants are explicit, making the algorithms easier to understand and verify. Indeed, this helped us discover improvements to some of the algorithms, both for correctness and for optimizations, and also exposed some remaining correctness concerns.

2 Language and case studies

We use a very high level, executable language, called DistAlgo, that has an operational semantics [23]. We use parts of two case studies as examples in describing our method.

Language. To support distributed programming at a high level, we add four main concepts to commonly used object-oriented programming languages, such as Java and Python: (1) processes as objects, and sending of messages, (2) yield points and waits for control flows, and handling of received messages, (3) computations using high-level queries and message history sequences, and (4) configuration of processes and communication mechanisms. The following paragraphs describe the constructs that support these concepts in DistAlgo. For other constructs, we mostly use Python syntax (indentation for scoping, '`:`' for separation, '`#`' for comments, etc.), for succinctness, except with a few conventions from Java. The `skip` statement does nothing. We adopt the convention that any method named `setup` implicitly assigns each of its parameters to a field with the same name as the parameter before executing the rest of its body.

Processes and sending of messages. Process definition is done by defining classes that extend a special class `Process`. This is analogous to thread definition in Java and Python, which is done by defining classes that extend a special class `Thread`. The class must define a `run` method. The `start` method inherited from `Process` starts the execution of the process, which executes its `run` method. Processes can be created using constructors of process classes. Those constructors have an optional additional parameter that specifies the site (machine) on which the new process should be created. Processes can also be created by calling `newprocesses(n,P,s)`, which creates and returns a set of `n` processes of class `P` on site `s`.

A send-statement `send m to p` sends a message `m` to a process `p`. If `p` is a set of processes, `m` is sent to each process in the set. A message can be a tuple, where the first component is a string specifying the kind of the message.

Control flows and handling of received messages. The key idea is to use labels to specify program points where control flow can yield to handling of messages and resume afterwards. A yield point is a statement of the form `-- l`, where `l` is a label that names this point in the program. Messages are handled only at yield points, so code segments not containing yield points are atomic. Handling of received messages is expressed using receive-definitions, which are members of class definitions for processes and are of the form:

$$\texttt{receive m}_1\texttt{ from p}_1\texttt{,...,m}_k\texttt{ from p}_k\texttt{ at l}_1\texttt{,...,l}_j\texttt{: stmt}$$

where each $\texttt{m}_i$ is a variable or tuple pattern. This allows messages that match any one of $\texttt{m}_1\texttt{ from p}_1\texttt{, ..., m}_k\texttt{ from p}_k$ to be handled at yield points labeled any one of $\texttt{l}_1\texttt{,...,l}_j$, by executing the statement `stmt` at those points. A tuple pattern is a tuple in which each component is a constant, a variable possibly prefixed with "`=`", or a wildcard. A variable prefixed with "`=`" means that the corresponding part of the tuple being matched must equal the value of the variable for pattern matching to succeed. A variable that is not prefixed with "`=`" matches any value and gets bound to the corresponding part of the tuple being matched. A wildcard, written as "`_`", matches any value. The at-clause is optional, and the default means all yield points. The from-clause is also optional. As syntactic sugar, a receive-definition used at only one yield point can be written at that point.

Synchronization uses the await-statement, whose general form is

$$\texttt{await bexp}_1\texttt{: stmt}_1\texttt{ or ... or bexp}_k\texttt{: stmt}_k\texttt{ timeout t: stmt}$$

This statement waits for one of the Boolean expressions $\texttt{bexp}_i$ to become true or until `t` seconds have passed and then executes the corresponding statement. The statements $\texttt{stmt}_i$ and the timeout-clause are optional. An await-statement must be preceded by a yield point; if a yield point is not specified explicitly, the default is that all message handlers can be executed at this point.

High-level queries. Synchronization conditions can be expressed using high-level queries—quantifications, comprehensions, and aggregates—over sets of processes and sequences of messages. We define operations on sets; operations on sequences are the same except that elements are processed in order, and square brackets are used in place of curly braces.

- Quantifications are of the following two forms. Each variable $\texttt{v}_i$ enumerates elements of the set value of expression $\texttt{exp}_i$; the return value is whether, for each or some, respectively, combination of values of $\texttt{v}_1\texttt{,...,v}_k$, the value of Boolean expression `bexp` is `true`.

  $$\texttt{each v}_1\texttt{ in exp}_1\texttt{, ..., v}_k\texttt{ in exp}_k\texttt{ | bexp}$$
  $$\texttt{some v}_1\texttt{ in exp}_1\texttt{, ..., v}_k\texttt{ in exp}_k\texttt{ | bexp}$$

- Comprehensions are of the following form. Each variable $\texttt{v}_i$ enumerates elements of the set value of expression $\texttt{exp}_i$; for each combination of values of $\texttt{v}_1\texttt{,...v}_k$, if the value of Boolean expression `bexp` is `true`, the value of expression `exp` forms an element of the resulting set.

  $$\texttt{\{exp: v}_1\texttt{ in exp}_1\texttt{, ..., v}_k\texttt{ in exp}_k\texttt{ | bexp\}}$$

We abbreviate `{v: v in exp | bexp}` as `{v in exp | bexp}`.

- Aggregates are of the form `agg(exp)`, where `agg` is an operation, such as `count` or `min`, specifying the kind of aggregation over the set value of `exp`.
- In the query forms above, each $\texttt{v}_\texttt{i}$ can also be a tuple pattern, in which case each enumerated element of the set value of $\texttt{exp}_\texttt{i}$ is matched against the pattern before `bexp` is evaluated. We omit `|bexp` when `bexp` is `true`.

We use `{}` for empty set; `s.add(x)` and `s.del(x)` for element addition and deletion, respectively; and `x in s` and `x not in s` for membership test and its negation, respectively. We overload `or` to work for sets; $\texttt{s}_1$ `or` $\texttt{s}_2$ returns $\texttt{s}_1$ if $\texttt{s}_1$ is non-empty, otherwise it returns $\texttt{s}_2$.

DistAlgo has two built-in sequences, `received` and `sent`, containing all messages received and sent, respectively, by a process.

- Sequence `received` is updated only at yield points. An arrived message `m` for which the program contains a matching receive-definition is added to `received` when the program reaches a yield point where `m` is handled, and all matching message handlers associated with that yield point are executed for `m`. An arrived message for which the program contains no matching receive-definitions is added to `received` at the next yield point. The sequence `sent` is updated at each send-statement.
- `received(m from p)` is a shorthand for `m from p in received`; `from p` is optional, but when it is used, each message in `received` is automatically associated with the corresponding sender. `sent(m to p)` is a shorthand for `m to p in sent`; `to p` is optional, but when it is used, `p` is the process or set of processes in the corresponding send-statement.

Configuration. Configuration statements can specify various aspects of configuration. For example, `use fifo_channel` and `use reliable_channel` specify that channels are required to be FIFO and reliable, respectively; by default, channels are not required to be FIFO or reliable. The configuration statement `use Lamport_clock` specifies that Lamport logical clock [15, 9, 25] is used; this configures sending and receiving of messages to update the clock, and defines a function `Lamport_clock()` that returns the value of the clock.

Case studies. We use parts of two important algorithms as case studies: (1) van Renesse's pseudo code for multi-Paxos for distributed consensus [33], which has been worked on for a long time, with the pseudo code remaining the same for a year or more, and is in the process of being made a technical report, and (2) Lamport's description of distributed mutual exclusion algorithm [15], which Lamport developed to illustrate the logical clock he invented. We use them because they are the clearest descriptions we found for these problems.

van Renesse's pseudo code for multi-Paxos is for a set of leaders, commanders, scouts, and acceptors to reach consensus among a set of replicas in serving a sequence of requests from clients. A replica receives client requests and proposes to leaders, and receives decisions from leaders and replies back to clients; a leader spawns off commander and scouts to do the two phases of the consensus algorithm; commander and scouts communicates with acceptors to try to have proposed values accepted.

Lamport's distributed mutual exclusion is for a set of processes accessing a shared resource that can only be used by one process at a time. A process maintains a queue of pending requests sorted by their logical timestamps, adds self to the request queue and sends a message to all others to request the resource, waits for all others to reply and for self to be first on the queue to get access, and sends release messages to all and dequeues itself afterwards; it enqueues any request upon receiving the request message, and dequeues it upon receiving the release message.

3 High-level specifications of distributed algorithms

Our method aims to specify distributed algorithms at a high level while keeping them fully executable as they are designed for. The key idea is to preserve the sending and receive of messages while abstracting away details of local computations.

Abstractions for specifying distributed algorithms. Our method exploits two basic abstractions—message history sequences and queries over sets and sequences—and has four main components:

1. abstracting waiting on received messages using high-level synchronization with explicit wait,
2. abstracting when to send messages using high-level assertions over sets and sequences,
3. abstracting what to send in messages to whom using high-level set and aggregate computations, and
4. abstracting what messages to send collectively using loops and high-level queries.

These abstractions help make invariants maintained in distributed algorithms explicit, and thus help make the algorithms easier to understand and to verify. Note that our method does not yet make all invariants explicit, if that is possible.

The method emphasizes sending of messages and synchronization, because a process has no control over when it receives what messages from whom, but only when and how to handle them once they arrive, and handling of received messages is driven by the need to send messages, besides waiting and yielding. Therefore, handling is implied by the four components above, especially as they all heavily use queries over received messages.

Message sequences. For a distributed process to make decisions, the key input is the history of messages it has sent and received. Therefore, at a high level, these decisions should be expressed in terms of the sequences of messages sent and received, not lower-level local updates after each message is sent or received.

High-level queries. Because distributed computations involve sets of processes and sequences of message, decision making mainly involves assertions and other computations over sets and sequences. To specify these assertions and computations at a high level, our method uses queries extensively, including logic quantifications, set comprehensions, and aggregate computations.

Overall method. The four components of our method are orthogonal and can be applied independently. We describe these components in more detail in four subsections and show precisely how they help specify distributed algorithms at a higher level.

Incremental computations. Although abstractions with high-level queries help make algorithms easier to understand and to verify, computations using these abstractions can be extremely inefficient, because they involve iteration over sets and sequences, and they are performed repeatedly as the sets and sequences are updated. This can take asymptotically much more time than necessary, and furthermore the space usage may be unbounded if the history of messages sent and received is used in actual implementations.

Optimization by incrementalization, e.g., [28, 12, 22, 21], transforms such expensive computations into efficient incremental maintenance of appropriate auxiliary values as the sets and sequences are updated. For distributed algorithms, the resulting incremental computations become efficient message handlers [23]. In fact, it was during the study of such optimizations in the last several years that we developed the abstractions, which we believe was instrumental in leading us to discover improvements to some of the algorithms.

3.1 Explicit high-level synchronization

Synchronization is at the core of distributed systems. It requires waiting for certain conditions to become true before taking the corresponding actions. Because message passing is generally asynchronous in distributed systems, synchronization must be achieved by explicitly tracking synchronization conditions, maintaining their truth values as messages are received, until the conditions become true, and then taking the corresponding actions.

Expressing such synchronization at a low level requires, in general, sophisticated updates driven by the events of different kinds of messages being received, making it difficult to understand and verify the conditions that the process is waiting for.

We use three principles in specifying such synchronization at a high level: (1) specify the waiting on the conditions and corresponding actions explicitly using await-statements, (2) express the conditions using high-level queries over sequences of messages sent and received, and (3) minimize local updates in the actions.

Example. In multi-Paxos [33], a commander process is spawned by a leader for each adopted triple of ballot number, slot number, and proposal, to try to have it accepted by acceptors and notify replicas of the decisions, and in case of being preempted by a different ballot number, to notify the leader.

Fig. 1 shows the pseudo code for a commander in multi-Paxos. A commander maintains `waitfor`—the set of acceptors from which it waits for `p2b` messages.

It sends a `p2a` message to all acceptors and then handles each `p2b` message it receives from an acceptor, maintaining `waitfor` in one of two cases. When `|waitfor|<|acceptors|/2` in the first case, it sends a `decision` message to all replicas and exits; it sends a `preempted` message in the second case.

```
process $Commander(\lambda, acceptors, replicas, \langle b, s, p\rangle)$
  var $waitfor := acceptors$;

  $\forall\alpha \in acceptors : send(\alpha, \langle \mathbf{p2a}, self(), \langle b, s, p\rangle\rangle)$;
  for ever
    switch $receive()$
      case $\langle \mathbf{p2b}, \alpha, b'\rangle$ :
        if $b' = b$ then
          $waitfor := waitfor - \{\alpha\}$;
          if $|waitfor| < |acceptors|/2$ then
            $\forall\rho \in replicas$ :
              $send(\rho, \langle \mathbf{decision}, s, p\rangle)$;
            $exit()$;
          end if;
        else
          $send(\lambda, \langle \mathbf{preempted}, b'\rangle)$;
          $exit()$;
        end if;
      end case
    end switch
  end for
end process
```

Fig. 1. Pseudo code for a commander in multi-Paxos [33].

We specify a commander at a high level as follows. First, we specify the synchronization explicitly using an await-statement. Then, we note that `waitfor` can be queried from the set of `p2b` messages received and the given set of acceptors, so we do not maintain `waitfor` explicitly; instead of starting from all acceptors and removing certain acceptors until a minority remain, we directly check whether those certain acceptors are a majority. Finally, the corresponding actions are simply single send-actions, yielding the specification in Fig. 2.

The result is that the flow that leads to each send-action is made clearer, and the conditions for the actions can easily be read off. Similar improvements can be made to the specification of a scout.

```
class Commander extends Process:
   def setup(leader, acceptors, replicas, b, s, p): skip

   def run():
      send ('p2a', b, s, p) to acceptors
      await count({a: received(('p2b', =b) from a)}) > count(acceptors)/2:
         send ('decision', s, p) to replicas
      or received('p2b', b2) and b2!=b:
         send ('preempted', b2) to leader
```

Fig. 2. Higher-level specification for a commander in multi-Paxos.

3.2 Direct high-level assertions

Determining the state of a distributed system is key to synchronization and to making decisions in general. Because there is no shared memory, a process must assert the state to the best of its knowledge through sending and receiving messages. The truth values of assertions about the state must be updated as messages are sent and received.

We express assertions using high-level queries over sequences of messages sent and received, as for synchronization conditions. The queries may be in the forms of quantifications, comprehensions, and aggregates. However, a same assertion may be expressed using different forms of queries. Because quantifications are usually not supported in executable languages, loops and low-level updates are most often used. Even in many high-level specifications, comprehensions and aggregates are often used in place of quantifications; this can be error-prone or lead to poor performance.

For example, an existential quantification may be specified indirectly as a set comprehension followed by an emptiness test, but this may incur unnecessary space for maintaining the intermediate set. For another example, a universal quantification asserting that a number is greater than all elements in a set may be specified indirectly as the number being greater than the maximum element in the set, but this causes an error when the set is empty; a special boundary value may be used in case the set is empty, but this is error-prone and may be sensitive to the maximum or minimum number that can be represented, which may be determined by the memory word size.

Our core principle in specifying assertions at a high level is to express existentially and universally quantified properties directly using logic quantifications, not indirectly using aggregates or comprehensions. Quantifications are easier and clearer for correctly stating the requirements, and can be systematically converted to aggregates and comprehensions that allow the best optimizations [23].

Example. In Lamport's distributed mutual exclusion [15], a process that requests a resource at time `c` needs to wait for the following two key conditions to hold before it is granted the resource:

(i) the request time `(c,self)` in its request queue is ordered before every other request in the queue, and (ii) it has received an acknowledgment message from every other process timestamped later than `c`.

We express the assertion directly using three quantifications, including a nested quantification in the second condition. The result is that the conditions can be directly read off the assertion.

```
each (’request’,c2,p2) in q | (c2,p2)!=(c,self) implies (c,self) < (c2,p2)
and each p2 in s | some (’ack’, c2, =p2) in received | c2 > c
```

3.3 Straightforward high-level computations

A distributed algorithm is designed for a set of processes to achieve a goal via sending and receiving messages. Computations needed for achieving the goal generally involve various collections of processes and messages. This means that the algorithm specification must capture the effects of sending and receiving messages on the needed computations.

Expressing these computations at a low level requires explicitly storing the results of these computations and updating their values appropriately as relevant messages are sent and received. Maintaining these low-level values correctly through updates can be challenging and error-prone; some of them require combinations of sophisticated data structures, while others are tedious.

We use three principles in specifying such computations at a high level: (1) specify computations of aggregate values using aggregate queries over message sequences, (2) specify computations of set values using comprehensions over message sequences, and (3) specify repeated computations straightforwardly where the results are used.

Example. In multi-Paxos [33], an acceptor process responds to `p1a` messages from scouts with `p1b` messages in the first phase, and responds to `p2a` messages from commanders with `p2b` messages in the second phase.

```
process $Acceptor()$
  var $ballot\_num := \bot$, $accepted := \emptyset$;

  for ever
    switch $receive()$
      case $\langle \mathbf{p1a}, \lambda, b \rangle$ :
        if $b > ballot\_num$ then
          $ballot\_num := b$;
        end if;
        $send(\lambda, \langle \mathbf{p1b}, self(), ballot\_num, accepted \rangle)$;
      end case
      case $\langle \mathbf{p2a}, \lambda, \langle b, s, p \rangle \rangle$ :
        if $b \geq ballot\_num$ then
          $ballot\_num := b$;
          $accepted := accepted \cup \{\langle b, s, p \rangle\}$;
        end if
        $send(\lambda, \langle \mathbf{p2b}, self(), ballot\_num \rangle)$;
      end case
    end switch
  end for
end process
```

Fig. 3. Pseudo code for an acceptor in multi-Paxos [33].

Fig. 3 shows the pseudo code for an acceptor in multi-Paxos. An acceptor maintains `ballot_num`—a ballot number, and `accepted`—a set of accepted triples of ballot number, slot number, and proposal. It handles a `p1a` message by updating `ballot_num` and replying with a `p1b` message containing `ballot_num` and `accepted`, and handles a `p2a` message by updating `ballot_num` and `accepted` and replying with a `p2a` message containing `ballot_num`.

We specify an acceptor at a high level as follows. First, we note that `ballot_num` is updated to be the maximum from `p1a` and `p2a` messages, so we compute it using an aggregate. Then, we compute it straightforwardly where it is used in message handlers, yielding the specification in Fig. 4.

```
class Acceptor extends Process:
  def setup(): self.accepted = {}

  def run(): await false

  receive m:
    self.ballot_num = max({b: received('p1a',b)}+{b: received('p2a',b,_,_)} or {(-1,-1)})

  receive ('p1a', _) from scout:
    send ('p1b', ballot_num, accepted) to scout

  receive ('p2a', b, s, p) from commander:
    if b == ballot_num: accepted.add((b,s,p))
    send ('p2b', ballot_num) to commander
```

Fig. 4. Higher-level specification for an acceptor in multi-Paxos.

The result is that the invariants relating the sent messages to the received messages are made clearer. In particular, it allowed us to make explicit the property that `(b,s,p)` is added to `accepted` only if `b` equals `ballot_num`.

3.4 Collective send-actions

```
process $Replica(leaders, initial\_state)$
  var $state := initial\_state,\ slot\_num := 1$;
  var $proposals := \emptyset,\ decisions := \emptyset$;

  function $propose(p)$
    if $\nexists s : \langle s, p \rangle \in decisions$ then
      $s' := \min\{s \mid s \in \mathbb{N}^+ \wedge$
            $\nexists p' : \langle s, p' \rangle \in proposals \cup decisions\}$;
      $proposals := proposals \cup \{\langle s', p \rangle\}$;
      $\forall \lambda \in leaders : send(\lambda, \langle \mathbf{propose}, s', p \rangle)$;
    end if
  end function

  function $perform(\langle \kappa, cid, op \rangle)$
    if $\exists s : s < slot\_num \wedge$
              $\langle s, \langle \kappa, cid, op \rangle \rangle \in decisions$ then
      $slot\_num := slot\_num + 1$;
    else
      $\langle next, result \rangle := op(state)$;
      atomic
        $state := next$;
        $slot\_num := slot\_num + 1$;
      end atomic
      $send(\kappa, \langle \mathbf{response}, cid, result \rangle)$;
    end if
  end function

  for ever
    switch $receive()$
      case $\langle \mathbf{request}, p \rangle$ :
        $propose(p)$;
      case $\langle \mathbf{decision}, s, p \rangle$ :
        $decisions := decisions \cup \{\langle s, p \rangle\}$;
        while $\exists p' : \langle slot\_num, p' \rangle \in decisions$ do
          if $\exists p'' : \langle slot\_num, p'' \rangle \in proposals \wedge$
                    $p'' \neq p'$ then
            $propose(p'')$;
          end if
          $perform(p')$;
        end while;
    end switch
  end for
end process
```

Fig. 5. Pseudo code for a replica in multi-Paxos [33].

Distributed algorithms generally involve sending and receiving collections of related messages. Precise specifications of distributed algorithms are commonly centered around handling of individual received messages. This lower-level model makes it harder than necessary to understand the overall working of the algorithms.

In contrast, a distributed algorithm can be viewed as driven by send-actions, because send-actions are observable externally, which then incur the needed computations. Thus, distributed algorithms may be expressed at a higher level by specifying send-actions collectively.

Our method aims to specify send-actions collectively in three steps: (1) identify the kinds of sent messages, (2) for each kind of sent messages, collect all situations in which messages of this kind are sent, and (3) express the collective situations using loops, choosing for-loops over while-loops if possible.

Example. In multi-Paxos [33], a replica process holds the state of the application; it handles requests of operations from clients and proposes them with minimum slot numbers to leaders, and it handles decisions of operations from leaders, applies the operations following the order of slot numbers, and sends the results to clients.

Fig. 5 shows the pseudo code for a replica in multi-Paxos. A replica maintains `state`—the state of the application, `slot_num`—a slot number for the next operation to be applied, `proposals`—the set of proposals it sent to leaders, and `decisions`—the set of decisions it received from leaders. It handles a `request` message by calling function propose. It handles a decision message by repeatedly checking decisions, re-proposing a proposal if overridden by a decision, and calling function perform. Function propose(p) checks that requested operation p is not in decisions, finds a minimum unused slot number for it, updates proposals, and sends a propose message. Function perform checks whether the operation in the argument is in decisions; if so, it only increments slot_num; otherwise, it applies the operation to state, atomically updates state and increments slot_num, and sends the result to the client.

We specify a replica process at a high level as follows. First, we identify the two send-actions as the driving goals of the process. Then, we collect all situations in which propose messages are sent: they are for all request messages received, including those already proposed but whose proposed slots are overridden by decisions. Here, we add details to replace the set of positive natural numbers $\mathbb{N}^+$ with the range of integers from 1 to the maximum of the slot numbers used plus 1. Finally, we collect all situations in which response messages are sent: they are for all decision messages received, applied in increasing order of slot numbers. Here we increment slot_num in both branches together, not worrying about breaking the atomic block, because the local updates are atomic by default without any yield point in between. We obtain the specification in Fig. 6.

```
class Replica extends Process:
  def setup(leaders, initial_state):
    self.state = initial_state
    self.slot_num = 1

  def run():
    while true:
      -- propose
      for ('request',p) in received:
        if each ('propose',s,=p) in sent | some received('decision',=s,p2) | p2!=p:
          s = min({s in 1.. max({s: sent('propose',s,_)}+{s: received('decision',s,_)})+1
                   | not (sent('propose',s,_) or received('decision',s,_))})
          send ('propose', s, p) to leaders
      -- perform
      while some ('decision', =slot_num, p) in received:
        if not some ('decision', s, =p) in received | s < slot_num:
          client, cmd_id, op = p
          state, result = op(state)
          send ('respond', cmd_id, result) to client
        slot_num += 1
```

Fig. 6. Higher-level specification for a replica in multi-Paxos.

4 Experiments

We experimented with specifying a variety of important distributed algorithms in DistAlgo, including the same algorithms specified at both high levels and low levels, and discovered improvements to some of the algorithms. We also implemented DistAlgo, as described in [23], by automatically generating Python code from DistAlgo specifications following the operational semantics, and we tested the invariants and performance by running the generated implementations on many inputs.

Algorithm specifications. Table 1 lists five algorithms with which we had the most interesting experiences. The last two columns show the sizes of DistAlgo specifications at a high level and sizes of DistAlgo specifications containing low-level incremental updates; for multi-Paxos in the last row, the second size is for a specification corresponding to the pseudo code in [33]. Each specification includes specification of a driver for configuring and running the algorithm.

These sizes are clearly smaller than specifications in other languages. For example, our high-level specification for La Paxos is 44 lines, compared with 83 lines of PlusCal [26], 145 lines of I/O automata [13], 230 lines of Overlog [27], and 157 lines of Bloom [29]. For multi-Paxos, our high-level specification is 86 lines, compared with 130 lines of pseudo code in [33], and about 3000 lines of Python in an implementation of that pseudo code [32].

Algorithm	Description	Spec size	Incr size
La mutex	Lamport's distributed mutual exclusion [15]	31	43
2P commit	Two-phase commit [11]	32	55
La Paxos	Lamport's Paxos for distributed consensus [16, 17]	44	59
CL Paxos	Castro-Liskov's Byzantine Paxos [5]	72	81
vR Paxos	van Renesse's pseudo code for multi-Paxos [33]	86	132

Table 1. Distributed algorithms and sizes of DistAlgo specifications (number of lines).

Improvements. We discovered improvements to some of the algorithms, as well as correctness and performance issues, explained below.

La mutex. Our method specifies the key synchronization conditions using quantifications directly, as discussed in Section 3.2. Transforming them into best forms of set and aggregate queries led to two discoveries: (1) Lamport's original algorithm can be simplified to not enqueue and dequeue a process's own request, and (2) a standard heap-like data structure for maintaining the minimum of all pending requests in O(log n) time per update can be removed, and the number of pending earlier requests can be maintained instead in O(1) time per update.

2P commit. Our method leads to a succinct specification of a coordinator process consisting mainly of 4 queries: 2 await-conditions, an if-condition, and a set comprehension. Even though the core algorithm does not specify timeout for the waits, the succinct specification makes it easy to see that allowing timeout of the first await-statement is safe, but allowing timeout of the second await-statement is not safe.

La Paxos and CL Paxos. Our method eventually led to specifications that use quantifications directly and cleanly, almost exactly as stated in the original informal algorithm descriptions. Our earlier versions used aggregates, and we discovered later that some of them were incorrect, while others needed to use special boundary values.

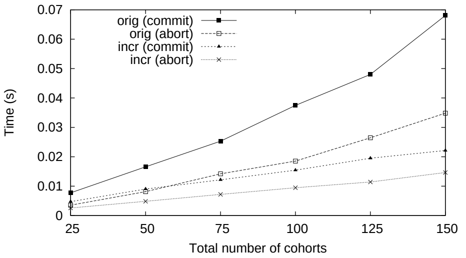
vR Paxos. Our method led to a specification easier to understand, as discussed in Sections 3.1, 3.3, and 3.4. The clearer specification led to two discoveries: (1) for a commander and scout, if the division operator `/`, which returns an integer in common programming languages, is used directly, the original checking of minority would be incorrect, and (2) for a replica, re-proposals, due to earlier proposals being overridden, are delayed unnecessarily.

Code generation. The table below shows the sizes (number of lines) of Python implementations generated from DistAlgo specifications, and the compilation time (ms) for generating the implementations. Our generated implementation of multi-Paxos corresponding to the pseudo code in [33] is 1099 lines of Python, much smaller than a manually written implementation of 3000 lines of Python [32]. Smaller higher-level specifications may take longer to compile than larger lower-level specifications, because transforming queries that use `received` and `sent` takes extra time, and may produce longer, more generic code.

We also measured time and space performance of generated implementations from both high-level and low-level DistAlgo specifications for these algorithms. The measurements confirmed the analyzed time and space complexities. The graph below shows the running times of generated implementations of 2P commit and 2P commit incr, for the commit case and abort case.

	Spec	Gen'd	Compil
Algorithm	size	size	time
La mutex	31	951	4.451
La mutex incr	43	960	4.988
2P commit	32	978	5.910
2P commit incr	55	1001	6.816
La Paxos	44	1003	9.121
La Paxos incr	59	999	7.613
CL Paxos	72	1044	13.055
CL Paxos incr	81	1024	12.348
vR Paxos	86	1116	19.064
vR Paxos incr	132	1099	21.602

"incr" indicates specifications containing low-level incremental updates.



Running times are averaged over 50 rounds and 15 independent runs, measured on an Intel Core-i7 2600K CPU with 16GB of main memory, running Linux 3.0.0 kernel and Python 3.2.2.

5 Related work

There has been much study on distributed algorithms, e.g., [30, 24, 10, 31], including especially much work on Paxos, from original [16], Byzantine [5, 20], made simple [17], made live in Google's Chubby service [6], and many more, to most recently precise pseudo code for full Paxos [33]. Distributed algorithms have been heavily and increasingly studied both because of their importance in increasingly more distributed applications, e.g., Google's computing infrastructure, and because of challenges in precisely specifying, implementing, and improving them to satisfy the needs of applications.

Distributed algorithms have been expressed in a wide range of languages and notations, from informal pseudo code to formal state machine based specifications, with many variations in between. Formal specification languages, such as I/O automata [24, 14], TLA+ [18], and PlusCal [19], are instrumental in precise verification. While study of languages is important, making specifications higher level is orthogonal, because the most essential language features are already present in many existing languages.

Besides state-machine based approaches, e.g., I/O automata [24, 14], established specification methods include notably the actor model [1] and general event-driven models where events include receipts of messages. These models focus on specifying actions and state transitions driven by the receipts of individual messages. Our specification method aims to make it easier to understand the algorithms at a high level, by abstracting away low-level state updates. It focuses on relating send-actions, which are externally observable, with the history of messages sent and received at a high level, by using high-level queries to express the assertions and computations.

More declarative languages for expressing distributed algorithms have also been studied, e.g., Datalog-based languages Overlog [2] and Bloom [3], and a logic-based language EventML [4, 7]. More declarative languages generally abstract away some or all control flow information and may be more succinct, but they are also harder to understand when used for specifications of algorithms, in which control flow is essential. Our method uses declarative queries over sets and sequences to express assertions and computations, and keeps the control flow of sending and receiving messages clear.

Our method can make the resulting executable specifications extremely inefficient if executed straightforwardly, because of repeated expensive high-level queries. Optimization by incrementalization [28, 12, 22, 21, 23] transforms such expensive queries into efficient incremental maintenance of appropriate auxiliary values. Invariants made explicit following our specification method not only help prove the correctness of the algorithms, but also help apply the optimizations. How to make more or all invariants explicit to make verification of distributed algorithms even easier is open for future study; so is the verification.

References

[1] G. Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, 1986.

[2] P. Alvaro, T. Condie, N. Conway, J. Hellerstein, and R. Sears. I do declare: Consensus in a logic language. *ACM SIGOPS Operating Systems Review*, 43(4):25–30, 2010.

[3] Berkeley Orders of Magnitude. Bloom Programming Language. http://www.bloom-lang.net/.

[4] M. Bickford. Component specification using event classes. In *Proceedings of the 12th International Symposium on Component-Based Software Engineering*, pages 140–155. Springer, 2009.

[5] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20:398–461, 2002.

[6] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live—An engineering perspective. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 398–407, 2007.

[7] CRASH Project. EventML. http://www.nuprl.org/software/#WhatisEventML, Last dated February 13, 2012.

[8] R. De Prisco, B. Lampson, and N. Lynch. Revisiting the Paxos algorithm. *Theoretical Computer Science*, 243:35–91, 2000.

[9] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, pages 56–66, 1988.

[10] V. K. Garg. *Elements of Distributed Computing*. Wiley, 2002.

[11] J. Gray. Notes on Data Base Operating Systems. In *Advanced Course: Operating Systems*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481, 1978.

[12] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 157–166, 1993.

[13] http://groups.csail.mit.edu/tds/ioa/distributions/IOA_Toolkit-tools.tar.gz. The Paxos code is under Examples/Paxos.

[14] D. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. *The Theory of Timed I/O Automata*. Morgan Claypool Publishers, 2nd edition, 2010.

[15] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, 1978.

[16] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[17] L. Lamport. Paxos made simple. *SIGACT News (Distributed Computing Column)*, 32(4):51–58, 2001.

[18] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[19] L. Lamport. The PlusCal algorithm language. In *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing*, pages 36–60, 2009.

[20] L. Lamport. Byzantizing Paxos by refinement. In *Proceedings of the 25th International Symposium on Distributed Computing*, pages 211–224. Springer, 2011.

[21] Y. A. Liu, M. Gorbovitski, and S. D. Stoller. A language and framework for invariant-driven transformations. In *Proceedings of the 8th International Conference on Generative Programming and Component Engineering*, pages 55–64, 2009.

[22] Y. A. Liu, S. D. Stoller, M. Gorbovitski, T. Rothamel, and Y. E. Liu. Incrementalization across object abstraction. In *Proceedings of the 20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 473–486, 2005.

[23] Y. A. Liu, S. D. Stoller, B. Lin, and M. Gorbovitski. From clarity to efficiency for distributed algorithms. In *Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2012.

[24] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.

[25] F. Mattern. Virtual time and global states of distributed systems. In *Proc. International Workshop on Parallel and Distributed Algorithms*, pages 120–131, 1989.

[26] Mechanically checked safety proof of a Byzantine Paxos algorithm. http://research.microsoft.com/en-us/um/people/lamport/tla/byzpaxos.html. Last modified 1 September 2011.

[27] https://svn.declarativity.net/overlog-paxos/src/olg/core/.

[28] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, 1982.

[29] https://github.com/bloom-lang/bud-sandbox/tree/master/paxos.

[30] M. Raynal. *Distributed Algorithms and Protocols*. Wiley, 1988.

[31] M. Raynal. *Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems*. Morgan & Claypool, 2010.

[32] E. G. Sirer, August 12, 2011. Email.

[33] R. van Renesse. Paxos made moderately complex, October 11, 2011. An online version is at www.cs.cornell.edu/courses/CS7412/2011sp/paxos.pdf.