# Invariants in Distributed Algorithms*

Yanhong A. Liu  Scott D. Stoller  Saksham Chand  Xuetian Weng
Computer Science Department, Stony Brook University, Stony Brook, NY 11794, USA
{liu,stoller,schand,xweng}@cs.stonybrook.edu

**Abstract**

We will discuss making invariants explicit in specification of distributed algorithms. Clearly this helps prove properties of distributed algorithms. More importantly, we show that this helps make it easier to express and to understand distributed algorithms at a high level, especially through direct uses of message histories. We will use example specifications in TLA+, for verification of Paxos using TLAPS, as well as complete executable specifications in DistAlgo, a high-level language for distributed algorithms.

## 1 Specification and verification of distributed algorithms

Distributed algorithms must deal with failures of processes and communication channels to provide correct distributed system functions. They can be subtle and difficult to understand even when they appear simple. Therefore, it is important to be able to specify these algorithms precisely, at a high level, to better understand them, and to execute them precisely and rigorously verify their correctness properties.

Many formal specification languages have been developed for specifying distributed algorithms, for example, IOA (for Input/Output Automata) [18, 4], TLA+ (for Temporal Logic of Actions) [8, 20], and PlusCal [9]. IOA and TLA+ are based on state machines, whereas PlusCal provides higher-level control flows.

TLA+ has the advantage that it needs only ordinary mathematics to specify system actions over states and only minimum temporal logic to describe temporal properties. While PlusCal supports higher-level control flows, its restriction on labels for expressing the grain of atomicity makes it less useful for distributed algorithms.

We show that even with state-machine based specifications, distributed algorithms can be expressed at a higher level, by using high-level queries over message history variables, capturing important invariants explicitly. We further show that higher-level control flows can be supported allowing any grain of atomicity in the specification.

We have developed a language, DistAlgo [16], that supports both and is directly executable. DistAlgo has a formal operational semantics and an efficient implementation extending Python. We have written corresponding specifications and abstract specifications in TLA+ for Multi-Paxos and variants and developed proofs of safety properties using the proof system TLAPS. We have also built automatic translators from DistAlgo to TLA+ for model checking by TLC.

*This work was supported in part by NSF under grants CCF-1414078, IIS-1447549, and CNS-1421893, and ONR under grant N000141512208.

## 2 A high-level language

For expressing distributed algorithms at a high level, DistAlgo [16] supports four main concepts by building on an object-oriented procedural language, such as Python: (1) distributed processes that can send messages, (2) control flow for handling received messages, (3) high-level queries for synchronization conditions, and (4) configuration for setting up and running. Appendix A shows Lamport's algorithm for distributed mutual exclusion in English and a high-level specification of it in a complete executable program in DistAlgo.

In particular, a `run` definition specifies the main flow of control of a process. Received messages are handled either synchronously, by querying `received` messages in `await` statements, or asynchronously, by running `receive` handlers at yield points. A yield point is labeled with `--` or is implicit before any `await` statement. A synchronization condition using message history variables is an explicit invariant asserting that the condition is true when passing that program point.

For efficient implementation, results of expensive queries are maintained incrementally as messages are sent and received [17, 12, 16]. High-level specifications and systematic incrementalization have allowed us to discover simplifications and improvements to some algorithms we studied [15, 16, 13].

## 3 Formal semantics

We give an abstract syntax and operational semantics for a core language for DistAlgo [16]. The operational semantics is a reduction semantics with evaluation contexts [23, 21]. The core language supports definition of classes containing methods and receive handlers. Expressions and statements include set comprehensions and quantifications with tuple patterns in membership clauses, object creation, method calls, `start` (for starting process), `send` (for sending messages), and assignments to local variables and object fields. Control structures include `await` statement, `if` statement, `while` loops, `for` loops that iterate over sets, and sequential composition. Processes are objects, so object creation subsumes dynamic process creation.

A state consists of the local state of each process plus the contents of the communication channels between each pair of processes. The local state of a process consists mainly of a heap and a statement representing the remaining code to be executed by that process. Intuitively, a transition removes the part of the statement that was just executed and updates the contents of the heap and message channels appropriately. For example, if the remaining code for process $P_1$ is `x:=1; y:=2`, then a transition by $P_1$ leads to a state in which `x` equals `1` and the remaining code for $P_1$ is `y:=2`. Evaluation contexts are used to identify the sub-expression or sub-statement of the remaining code to be evaluated next. For example, for `if` statements, the evaluation context indicates that the condition expression should be evaluated next, unless it has already been evaluated to a Boolean literal. Loops are handled by unrolling. Method calls are handled by inlining the statement produced by substituting the arguments for the parameters in the method body.

An execution is a sequence of transitions starting from an initial state. An execution may terminate (i.e., there is no remaining code for any process to execute), get stuck (e.g., due to an unsatisfied `await` statement), or run forever due to an infinite loop or infinite recursion.

## 4 Proofs using TLAPS

We present our work [2, 1] on specification and verification of distributed algorithms in TLA$^+$ and TLAPS. As case study, we use Paxos [6]—the most famous algorithm for distributed consensus—and its variants. Building on Lamport et al.'s TLA$^+$ specification of Paxos [10], we specified Multi-Paxos in [2] and presented its formal proof of correctness written in and checked by TLAPS. Experiences gained from this work and the presence of high-level executable languages like DistAlgo, led us to explore a unique style of writing specifications in [1], using only message history variables. We observed that not using and maintaining other state variables yields simpler specifications that are more declarative and easier to understand. It also allows easier proofs to be developed by needing fewer invariants and facilitating proof derivations. We observed that the sizes of specifications and proofs were reduced by 25% and 27%, respectively, for Basic Paxos, and 46% (from about 100 lines to about 50 lines) and 48% (from about 1000 lines to about 500 lines), respectively, for Multi-Paxos. Overall we needed 54% fewer manually written invariants and our proofs had 46% fewer obligations.

We also wrote TLA+ specifications corresponding to complete DistAlgo programs for Multi-Paxos and variants and developed their safety proofs in TLAPS [13]. The specification and proof sizes are much larger. We are developing systematic methods to generate abstract specifications in DistAlgo from complete DistAlgo programs before translation to TLA+.

## 5 Translation to TLA+ and model checking using TLC

We built two automatic translators from DistAlgo to TLA+. The first one translates DistAlgo program in a straightforward method by lowering each DistAlgo statement to one or more corresponding TLA+ actions. It closely follows the semantics of DistAlgo and handles all DistAlgo features. It generates sub-optimal TLA+ specification [22]. In the second, improved translator, we introduced a lower-level intermediate representation and exploited the approaches used in traditional compilers, such as dead code elimination and constant propagation. This allowed us to generate much simpler specifications, comparable to hand-written TLA+ specification.

As case study, we translated Lamport's distributed mutual exclusion algorithm [5] in DistAlgo and compared the generated TLA+ specifications generated by our two translators with two hand-written TLA+ specifications. One is written in TLA+ by Lamport [7], and the other one is written in PlusCal by Merz [19]. We used TLC to check the safety property for all four specifications, with the same configuration (number of processes, etc.). The numbers of distinct states found by TLC are 28,358 for the TLA+ version by Lamport; 37,978 for the specification generated by the improved translator; 1,180,688 for the PlusCal version by Merz; and 2,052,276 for the specification generated by the first translator.

## References

[1] Saksham Chand and Yanhong A Liu. Simpler specifications and easier proofs of distributed algorithms using history variables. In *NASA Formal Methods Symposium*, pages 70–86. Springer, 2018.

[2] Saksham Chand, Yanhong A Liu, and Scott D Stoller. Formal verification of multi-paxos for distributed consensus. In *International Symposium on Formal Methods*, pages 119–136. Springer, 2016.

[3] Vijay K. Garg. *Elements of Distributed Computing.* Wiley, 2002.

[4] Stephen J. Garland and Nancy A. Lynch. Using I/O automata for developing distributed systems. In *Foundations of Component-Based Systems*, chapter 13, pages 285–312. Cambridge University Press, 2000.

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

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

[7] Leslie Lamport. Distributed algorithms in TLA+. PODC 2000 Tutorial `https://www.podc.org/podc2000/lamport.html`, 2000.

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

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

[10] Leslie Lamport, Stephan Merz, and Damien Doligez. Paxos.tla. `https://github.com/tlaplus/v1-tlapm/blob/master/examples/paxos/Paxos.tla`. Last modified Fri Nov 28 10:39:17 PST 2014 by Lamport. Accessed Feb 6, 2018.

[11] Bo Lin and Yanhong A. Liu. DistAlgo: A language for distributed algorithms. `http://github.com/DistAlgo`, 2017. Beta release September 27, 2014, latest release November 23, 2017.

[12] Yanhong A. Liu, Jon Brandvein, Scott D. Stoller, and Bo Lin. Demand-driven incremental object queries. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, pages 228–241. ACM Press, 2016.

[13] Yanhong A. Liu, Saksham Chand, and Scott D. Stoller. Moderately complex Paxos made simple: High-level specification of distributed algorithm. *Computing Research Repository*, arXiv:1704.00082 [cs.DC], Mar. 2017 (Revised July 2017). `http://arxiv.org/abs/1704.00082`.

[14] Yanhong A. Liu, Bo Lin, and Scott Stoller. DistAlgo Language Description. `distalgo.cs.stonybrook.edu`, 2017. Last revised March 24, 2017.

[15] Yanhong A. Liu, Scott D. Stoller, and Bo Lin. High-level executable specifications of distributed algorithms. In *Proceedings of the 14th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 95–110. Springer, 2012.

[16] Yanhong A. Liu, Scott D. Stoller, and Bo Lin. From clarity to efficiency for distributed algorithms. *ACM Transactions on Programming Languages and Systems*, 39(3):12:1–12:41, May 2017.

[17] Yanhong A. Liu, Scott D. Stoller, Bo Lin, and Michael Gorbovitski. From clarity to efficiency for distributed algorithms. In *Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 395–410, 2012.

[18] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.

[19] Stephan Merz. Lamport's algorithm, May 8 2010. Email communication with Yanhong Annie Liu.

[20] Microsoft Research. The TLA Toolbox. `http://lamport.azurewebsites.net/tla/toolbox.html`, Last modified January 30, 2018.

[21] Traian Florin Serbanuta, Grigore Rosu, and Jose Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*, 207:305–340, 2009.

[22] Xuetian Weng. Verification of distributed algorithms. Research proficiency exam report, Stony Brook University, Aug. (Revised Mar. 17, 2015) 2014.

[23] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

## A Lamport's algorithm for distributed mutual exclusion

Lamport developed an algorithm for distributed mutual exclusion to show the logical clock he invented [5].

The problem is that $n$ processes access a shared resource, and need to access it mutually exclusively, in what is called a critical section (CS), i.e., there can be at most one process in a critical section at a time. The processes have no shared memory, so they must communicate by sending and receiving messages. Lamport's algorithm assumes that communication channels are reliable and first-in-first-out (FIFO).

Figure 1-left shows Lamport's original description of the algorithm, except with the notation $<$ instead of $\longrightarrow$ in rule 5 (for comparing pairs of logical time and process id using lexical ordering: `(t,p)` $<$ `(t2,p2)` iff `t` $<$ `t2` or `t` $=$ `t2` and `p` $<$ `p2`) and with the word "acknowledgment" added in rule 5 (for simplicity when omitting a commonly omitted [18, 3] small optimization mentioned in a footnote). This description is the most authoritative, is at a high level, and uses the most precise English we found.

The algorithm is safe in that at most one process can be in a critical section at a time. It is live in that some process will be in a critical section if there are requests. It is fair in that requests are served in the order of the logical times (paired with process ids) of the request messages. Its message complexity is $3(n-1)$ in that $3(n-1)$ messages are required to serve each request.

Figure 1-right shows a high-level specification of Lamport's algorithm in a complete executable program in DistAlgo [14, 11]. It is obtained by first expressing Lamport's algorithm directly and then applying simplification and improvement enabled by systematic incrementalization to removed the need to maintain request queues [16].

The algorithm is then defined by the following five rules. For convenience, the actions defined by each rule are assumed to form a single event.

1. To request the resource, process $P_i$ sends the message $T_m$:$P_i$ *requests resource* to every other process, and puts that message on its request queue, where $T_m$ is the timestamp of the message.

2. When process $P_j$ receives the message $T_m$:$P_i$ *requests resource*, it places it on its request queue and sends a (timestamped) acknowledgment message to $P_i$.

3. To release the resource, process $P_i$ removes any $T_m$:$P_i$ *requests resource* message from its request queue and sends a (timestamped) $P_i$ *releases resource* message to every other process.

4. When process $P_j$ receives a $P_i$ *releases resource* message, it removes any $T_m$:$P_i$ *requests resource* message from its request queue.

5. Process $P_i$ is granted the resource when the following two conditions are satisfied: (i) There is a $T_m$:$P_i$ *requests resource* message in its request queue which is ordered before any other request in its queue by the relation $<$. (To define the relation $<$ for messages, we identify a message with the event of sending it.) (ii) $P_i$ has received an acknowledgment message from every other process timestamped later than $T_m$.

Note that conditions (i) and (ii) of rule 5 are tested locally by $P_i$.

```
process P:
  def setup(s):                # set of other procs
    self.s := s

  def mutex(task):             # do task in mutex
    -- request
    self.t := logical_time()                        #1
    send ("request", t, self) to s                  #1
    await each received("request",t2,p2) has        #5
           (not received("release",=t2,=p2)         #5
            implies (t,self) < (t2,p2))             #5
          and each p2 in s has                      #5
            some received("ack",t2,=p2) has t2>t    #5
    task()
    -- release
    send ("release", logical_time(), self) to s     #3

  receive ("request", _, p2):                       #2
    send ("ack", logical_time(), self) to p2        #2

  def run():                   # main flow of proc
    ...                        # other tasks of proc
    def task(): ...            # define some task
    mutex(task)                # do task with mutex
    ...                        # other tasks of proc
def main():                    # main of application
  ...                          # other tasks of appl
  configure channel = {reliable, fifo}
  configure clock = Lamport    # use Lamport clock
  ps := 50 new P               # create 50 P procs
  for p in ps: p.setup(ps-{p}) # pass in other procs
  for p in ps: p.start()       # start run of procs
  ...                          # other tasks of appl
```

Figure 1: Lamport's algorithm for distributed mutual exclusion.
Left: Lamport's original description in English. Right: Simplified algorithm (lines ended with a number in comment indicated by #) in a complete executable program in DistAlgo.