ApPLIED 2023

Specification and Runtime Checking of Derecho, A protocol for Fast Replication for Cloud Services

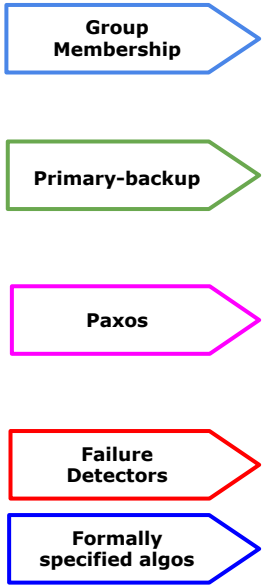
Kumar Shivam, Vishnu Paladugu, Yanhong A. Liu
Stony Brook University

Replication and consensus are important

Reliable distributed systems require replication and consensus among distributed processes to tolerate process and link failures

- Replication creates replicated processes to tolerate process failures
- Consensus makes a set of processes agree on a sequence of values—client operations—through message passing

Many algorithms for replication and consensus



	Name	Description	Language used
1	VS-ISIS	Reliable group communication, Birman-Joseph 1987 [7]	English (items)
2	VS-ISIS2	Virtual synchrony, Birman-Joseph 1987 [6]	English
3	EVS	Extended virtual synchrony for network partition, Amir et al 1995 [2, 3]	pseudocode
4	Paxos-VS	Virtually synchronous Paxos, Birman-Malkhi-van Renesse 2012 [8]	pseudocode
5	Derecho	Virtually synchronous state machine replication, Jha et al 2019 [29]	pseudocode
6	VR	Viewstamped replication, Oki-Liskov 1988 [55]	pseudocode (coarse)
7	VR-Revisit	VR revisited, Liskov 2012 [46]	English (items)
8	Paxos-Synod	Paxos in part-time parliament, Lamport 1998 [39]	TLA [38] (single-value)
9	Paxos-Basic	Single-value Paxos, Lamport 2001 [40]	English (items)
10	Paxos-Fast	Single-value Paxos with replicas proposing, Lamport 2006 [41]	English (items), TLA+ [18]
11	Paxos-Vertical	Single-value Paxos with external starting of leader election, Lamport-Malkhi-Zhou 2009 [44]	PlusCal [42]
12	CT	Single-value consensus with crash failures, Chandra-Toueg 1996 [13]	pseudocode
13	ACT	Single-value consensus in crash-recovery model, Aguilera-Chen-Toueg 2000 [1]	pseudocode
14	Paxos-Time	Paxos with time analysis, De Prisco-Lampson-Lynch 2001 [19]	IOA [49] (single-value)
15	Paxos-PVS	Single-value Paxos for proof, Kellomäki 2004 [35]	PVS [57]

* What's Live? Understanding Distributed Consensus

More examples of algorithms for replication and consensus

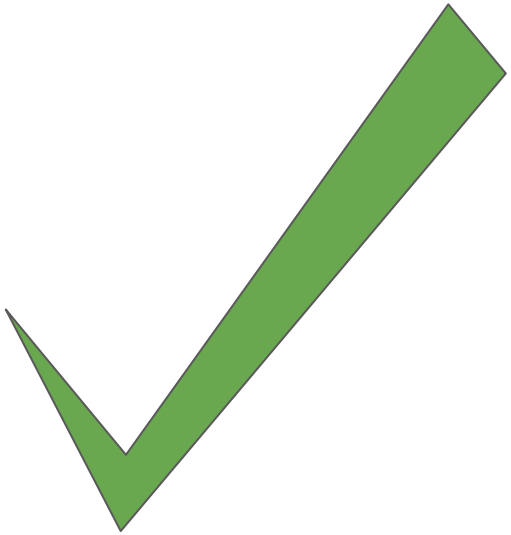
More variants

Formally specified algos

16	Chubby	Paxos in Google's Chubby lock service, Burrows 2006 [9]	English (partial items)
17	Chubby-Live	Chubby in Paxos made live, Chandra-Griesemer-Redstone 2007 [12]	English
18	Paxos-SB	Paxos for system builders, Kirsch-Amir 2008 [37]	pseudocode
19	Mencius	Paxos with leaders proposing in turn, Mao et al 2008 [51]	English (items)
20	Zab	Yahoo/Apache's Zookeeper atomic broadcast, Junqueira-Reed-Serafini 2011 [30]	English (items)
21	Zab-FLE	Zab with fast leader election, Medeiros 2012 [52]	pseudocode
22	EPaxos	Egalitarian Paxos, Moraru-Andersen-Kaminsky 2013 [53]	pseudocode
23	Raft	Consensus in RAMCloud, Ongaro-Ousterhout 2014 [56]	pseudocode
24	Paxos-Complex	Paxos made moderately complex, van Renesse-Altinbuken 2015 [62]	pseudocode, Python
25	Raft-Verdi	Raft for proof using Coq, Wilcox et al 2015 [64]	Verdi [64]
26	IronRSL	Paxos in Microsoft's IronFleet for proof, Hawblitzel et al 2015 [26]	Dafny [45]
27	Paxos-TLA	Paxos for proof using TLAPS, Chand-Liu-Stoller 2016 [11]	TLA+
28	LastVoting-PSync	Single-value Paxos in Heard-Of model for proof, Drăgoi-Henzinger-Zufferey 2016 [20]	PSync [20]
29	Paxos-EPR	Paxos in effectively propositional logic for proof, Padon et al 2017 [59]	Ivy [60]
30	Paxos-Decon	Paxos deconstructed, Garcia et al 2018 [24, 25]	Scala/Akka [28]
31	Paxos-High	Paxos in high-level executable specification, Liu-Chand-Stoller 2019 [47]	DistAlgo [48]

* What's Live? Understanding Distributed Consensus

So many algorithms!!! — challenge



Understanding

Correctness assurance

*Image by rawpixel.com on Freepik

This work

- develop a rigorous specification of Derecho replication protocol that corresponds closely to the pseudocode and is complete, precise, and directly executable
- discover and fix a number of issues in the Derecho pseudocode and helped improve the pseudocode
- demonstrate a practical method for developing a rigorous and improved spec through both manual inspection and automated runtime checking

These were enabled by DistAlgo language, compiler, and runtime checking framework

Bugs and fixes are checked and confirmed by the Derecho team. They also checked and confirmed that the bugs are not in their implementation in C++.

Outline

- Derecho overview
- DistAlgo and steps in developing the spec
- Precise specification
  - system state: shared state table (SST) & view
  - 2 parts: steady-state execution, view change: leader selection
- Systematic runtime checking
  - 3 key safety properties: validity, agreement, integrity
- Issues found and fixed in pseudocode: an example in leader selection

Derecho overview

- State machine replication using RDMA, enables direct access to remote memory without involving CPU
- Group multicast, within a group of member processes/nodes; supports:
  - Atomic Multicast: messages sent by a member node are either delivered to all member nodes or none at all
  - Total-ordered delivery: messages are delivered in the same order to all member nodes in the group
- Uses SST (Shared State Table), a distributed shared memory for sharing data and control information

Derecho overview - main parts

- Steady state execution
  - Atomic multicast delivery of client request across nodes
- View change
  - System progresses through series of view when members join or leave the group

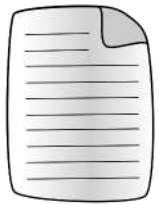
DistAlgo

- Language for distributed algorithms
- With a formal operational semantics
- Implemented by extending the Python compiler

Specification:

- High-level as pseudocode
- Precise with formal semantics
- Directly executable

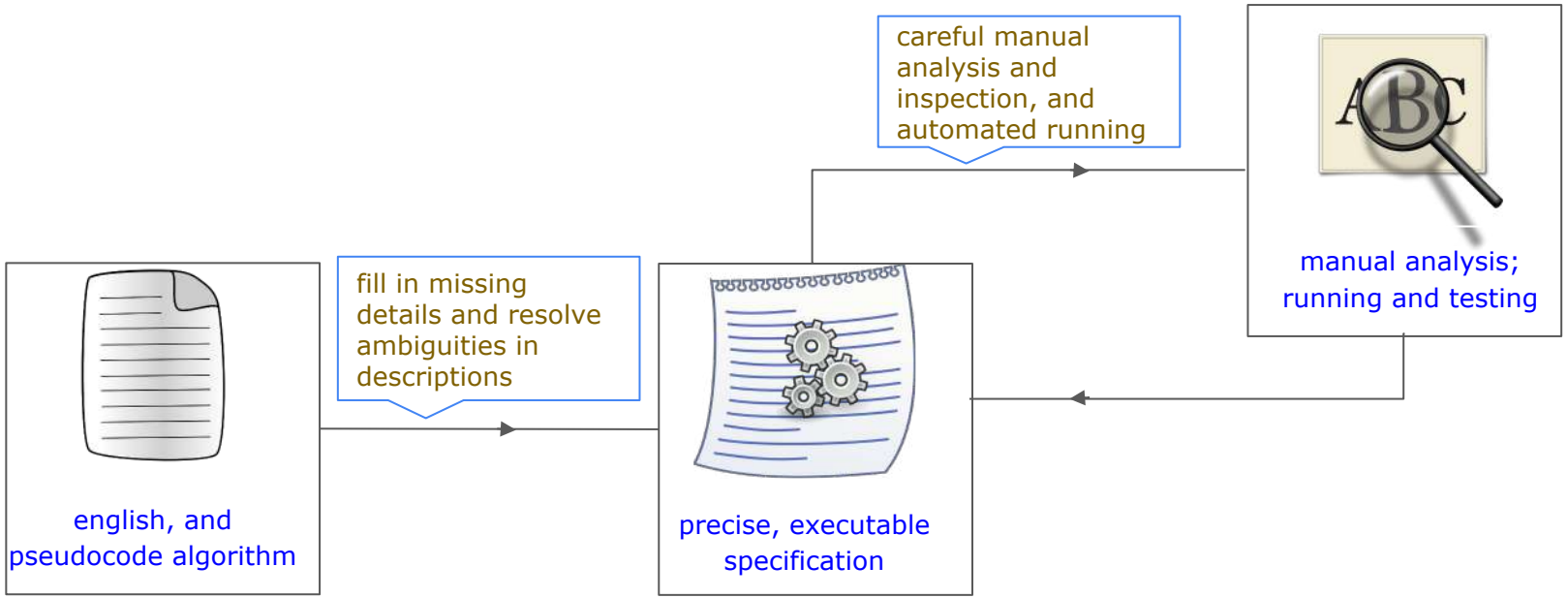
Steps in the development of spec



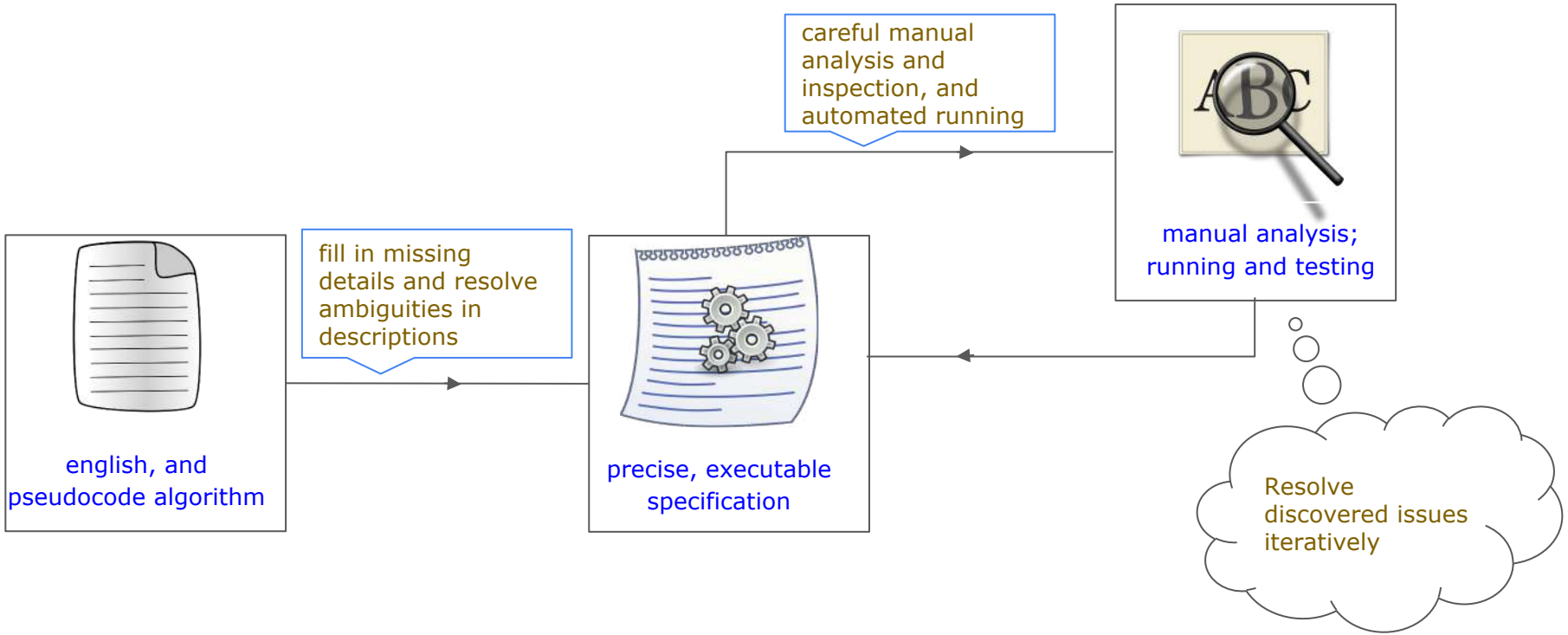
english, and
pseudocode algorithm

Steps in the development of spec

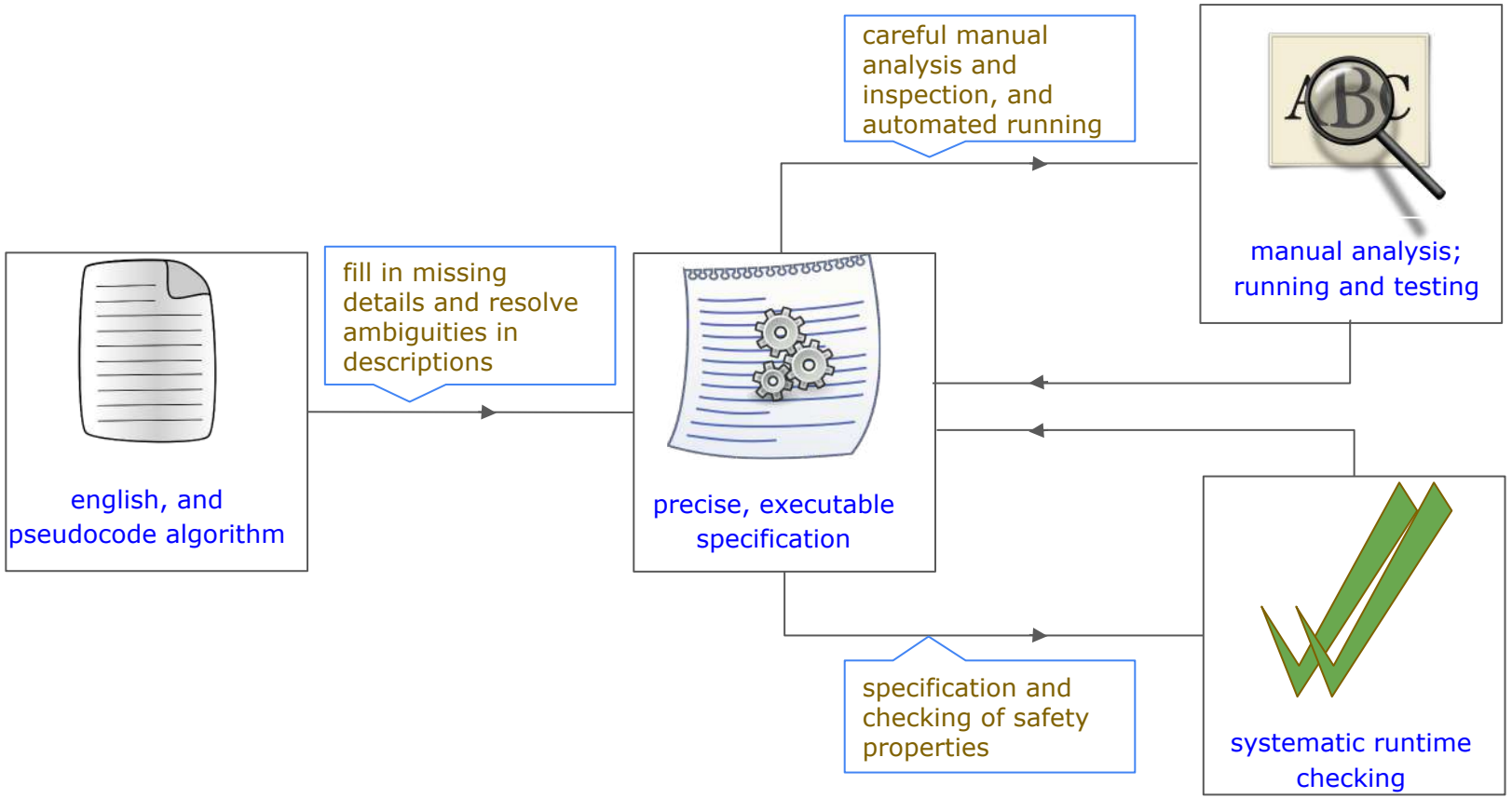
Steps in the development of spec



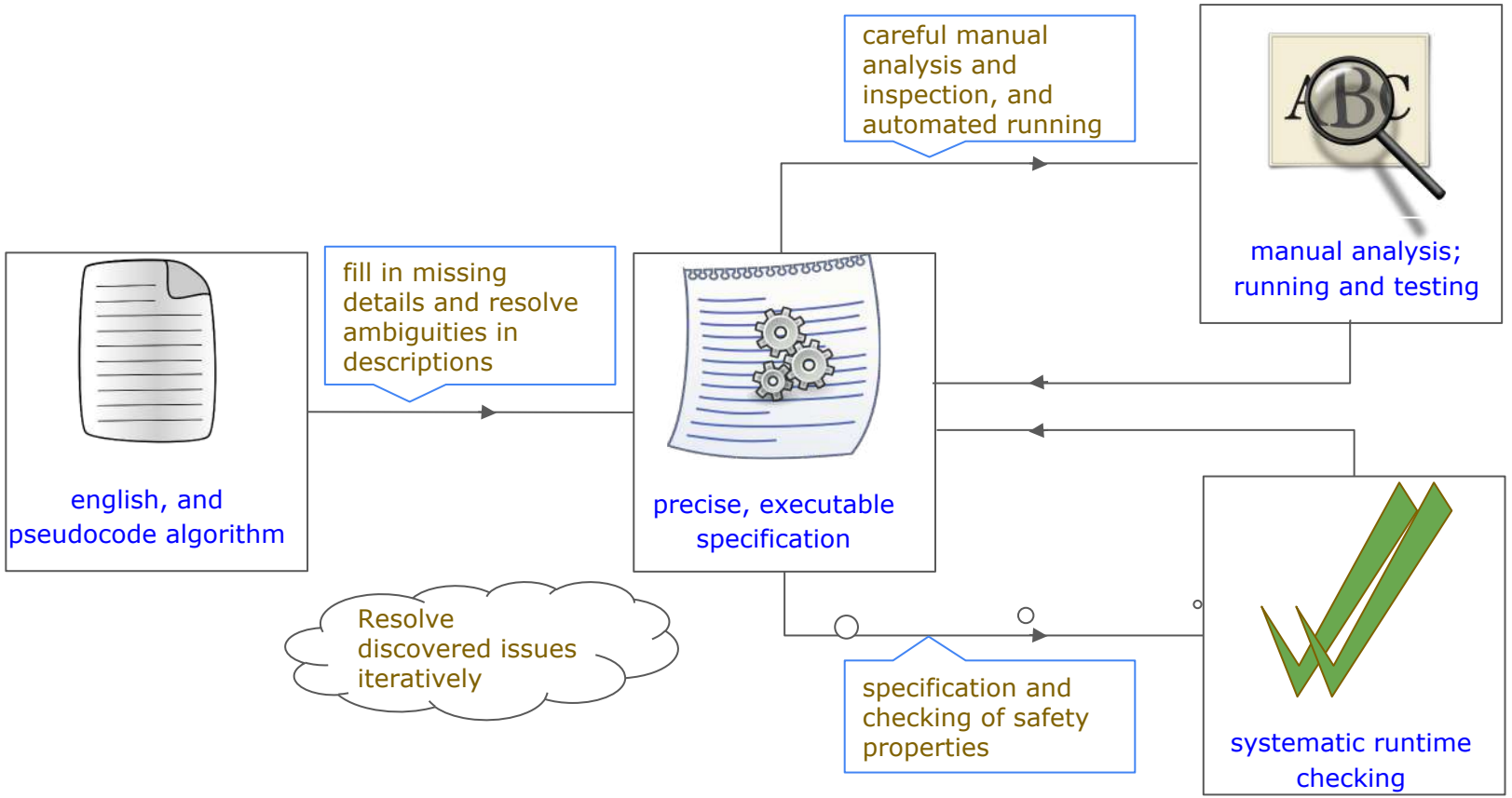
Steps in the development of spec



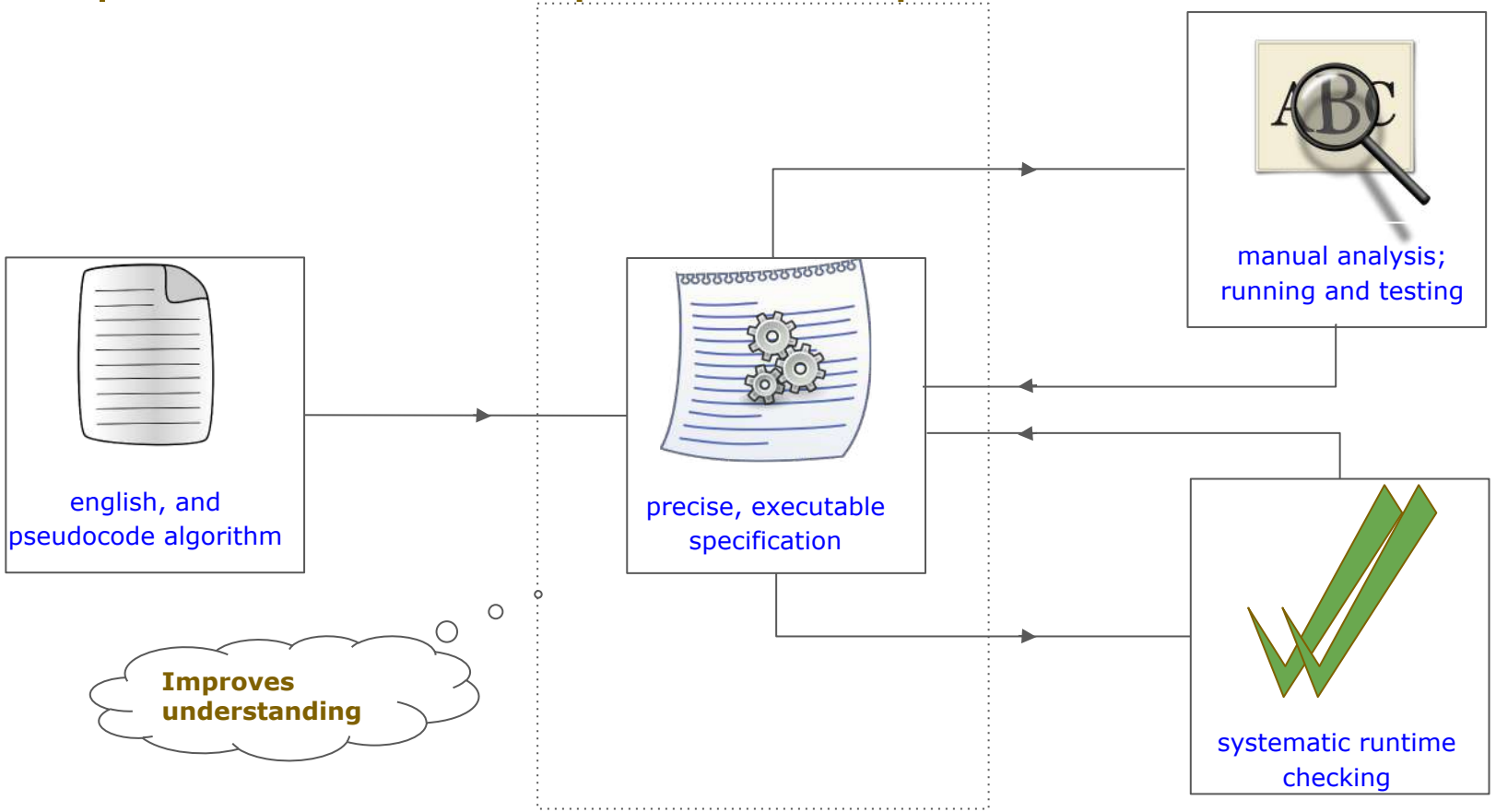
Steps in the development of spec



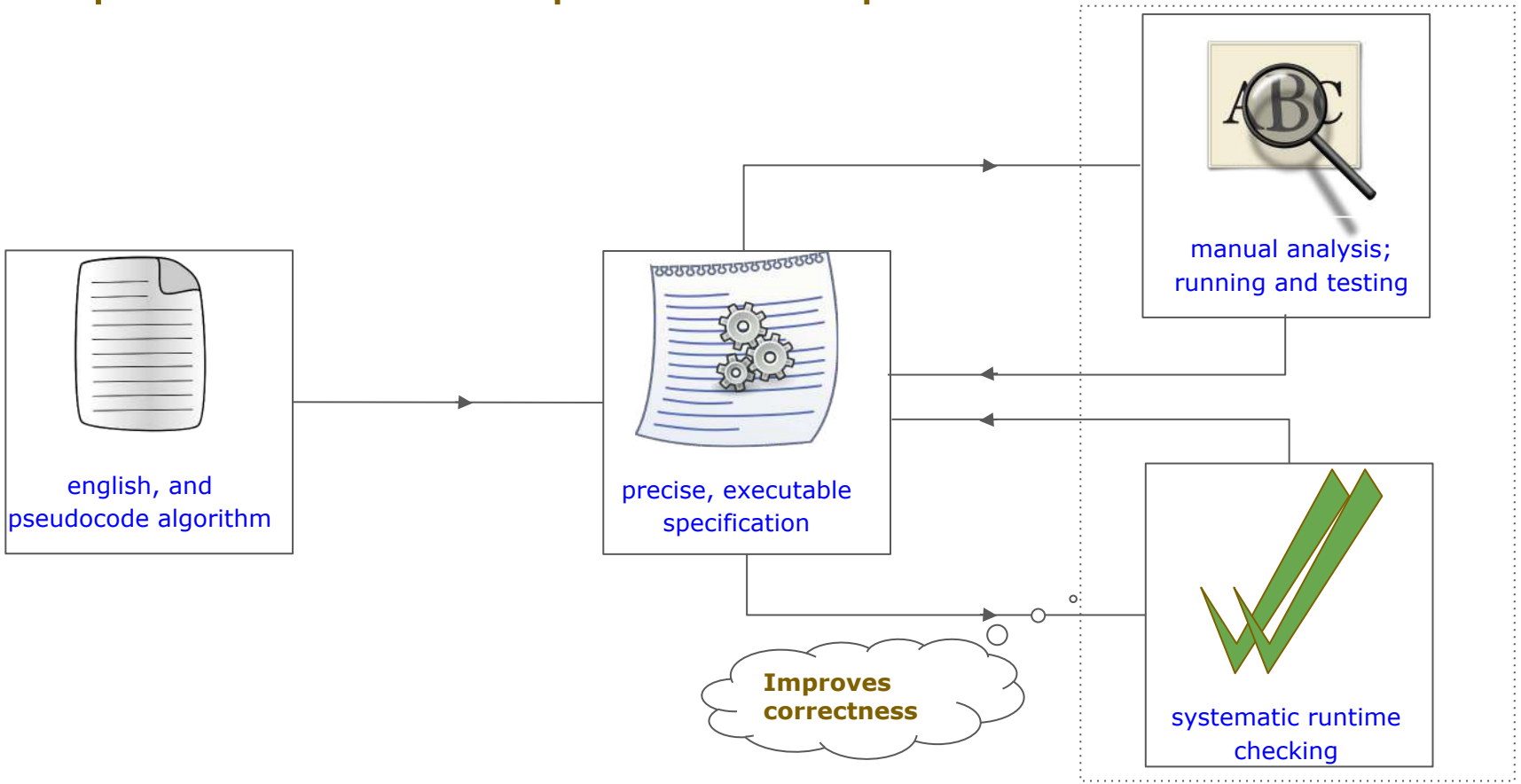
Steps in the development of spec



Steps in the development of spec



Steps in the development of spec

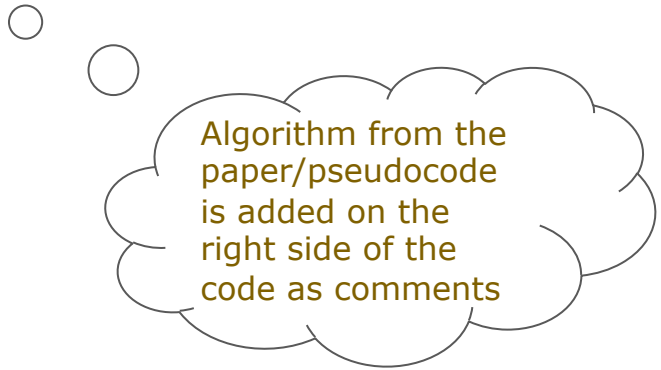


Precise specification

- Close correspondence to pseudocode
- Filled in missing details
- Ambiguities resolution

Three main parts:

- System state
- Steady state execution
- View change



Specifying system state

We define classes with fields that allow the algorithm steps in DistAlgo to match the corresponding steps in pseudocode exactly

- Shared state table (SST)
- View

Specifying system state: SST

```
class SSTRow:
  """A shared state table (SST) row that stores info about a node.
     A SST has a SSTRow for each member node and is stored in each member"""
  def __init__(self, n, window_size):  ## initialize SST columns for the row; all used in pseudocode but the last two
    # n:           ## number of member nodes in the group
    # window_size: ## length of vector of slots for storing client req msgs received by the node, directly or indirectly
    self.slots = [Slot() for _ in range(window_size)]  # (p.33) vector of window_size slots  ## for client request msgs
    self.received_num = [-1] * n                       # (p.33) number of messages received from each node  ### number-1
                                                         ## initialized in Node.init(), and set in receive_req()
    self.global_index = -1                 # ## global index of last message received from the most lagging node
    self.latest_delivered_index = -1       # ## min of self.global_index over all members
    self.latest_received_index = [-1] * n  # ## index of latest msg received from each node, set in recv to received_num
                                             ## i.e., self.received_num-1  ### redundant, but not clear with null msgs
    self.min_latest_received = [-1] * n    # ## for each node, min of latest_received_index over all rows in SST
    self.suspected = [False] * n           # ## for each node, whether that node is suspected to have failed
    self.wedged = False      # ## true when any node is suspected
    self.changes = []        # ## list of nodes suspected (or added from joins) to proposed as changes by the leader
    self.num_changes = 0     # ## number of nodes in self.changes, i.e., length of self.changes
    self.num_acked = 0       # ## number of nodes in self.changes acknowledged
                               ### set in 1 place by us, using num_changes
    self.num_committed = 0  # ## min of self.num_acked over not suspected nodes
    self.num_installed = 0  # ## number of nodes installed (added/removed) by the node, as proposed by the leader
    self.ragged_edge_computed = False  # ## true for leader calling terminate_epoch or others after leader did;
                                       ## the call happens when leader's num_committed > self's num_installed
    self.active = False  # (p.40) ## true when the epoch is active, only used at start  ### not in pseudocode
                         ### could use logical or over sst[my_rank].suspected or even just own suspected
```

SST is specified as a list of **SSTRow** objects

Specifying system state: SST

```
class SSTRow:
  """A shared state table (SST) row that stores info about a node.
     A SST has a SSTRow for each member node and is stored in each member"""
  def __init__(self, n, window_size):  ## initialize SST columns for the row; all used in pseudocode but the last two
    # n:           ## number of member nodes in the group
    # window_size: ## length of vector of slots for storing client req msgs received by the node, directly or indirectly
    self.slots = [Slot() for _ in range(window_size)]  # (p.33) vector of window_size slots  ## for client request msgs
    self.received_num = [-1] * n                       # (p.33) number of messages received from each node  ### number-1
                                                         ## initialized in Node.init(), and set in receive_req()
    self.global_index = -1                 # ## global index of last message received from the most lagging node
    self.latest_delivered_index = -1       # ## min of self.global_index over all members
    self.latest_received_index = [-1] * n  # ## index of latest msg received from each node, set in recv to received_num
                                             ## i.e., self.received_num-1  ### redundant, but not clear with null msgs
    self.min_latest_received = [-1] * n    # ## for each node, min of latest_received_index over all rows in SST
    self.suspected = [False] * n           # ## for each node, whether that node is suspected to have failed
    self.wedged = False      # ## true when any node is suspected
    self.changes = []        # ## list of nodes suspected (or added from joins) to proposed as changes by the leader
    self.num_changes = 0     # ## number of nodes in self.changes, i.e., length of self.changes
    self.num_acked = 0       # ## number of nodes in self.changes acknowledged
                               ### set in 1 place by us, using num_changes
    self.num_committed = 0  # ## min of self.num_acked over not suspected nodes
    self.num_installed = 0  # ## number of nodes installed (added/removed) by the node, as proposed by the leader
    self.ragged_edge_computed = False  # ## true for leader calling terminate_epoch or others after leader did;
                                       ## the call happens when leader's num_committed > self's num_installed
    self.active = False  # (p.40) ## true when the epoch is active, only used at start  ### not in pseudocode
                         ### could use logical or over sst[my_rank].suspected or even just own suspected
```

Definition of **SSTRow** with its fields

Specifying system state: SST

```
class SSTRow:
  """A shared state table (SST) row that stores info about a node.
     A SST has a SSTRow for each member node and is stored in each member"""
  def __init__(self, n, window_size):  ## initialize SST columns for the row; all used in pseudocode but the last two
    # n:           ## number of member nodes in the group
    # window_size: ## length of vector of slots for storing client req msgs received by the node, directly or indirectly
    self.slots = [Slot() for _ in range(window_size)]  # (p.33) vector of window_size slots  ## for client request msgs
    self.received_num = [-1] * n                       # (p.33) number of messages received from each node  ### number-1
                                                         ## initialized in Node.init(), and set in receive_req()
    self.global_index = -1                 # ## global index of last message received from the most lagging node
    self.latest_delivered_index = -1       # ## min of self.global_index over all members
    self.latest_received_index = [-1] * n  # ## index of latest msg received from each node, set in recv to received_num
                                             ## i.e., self.received_num-1  ### redundant, but not clear with null msgs
    self.min_latest_received = [-1] * n    # ## for each node, min of latest_received_index over all rows in SST
    self.suspected = [False] * n           # ## for each node, whether that node is suspected to have failed
    self.wedged = False      # ## true when any node is suspected
    self.changes = []        # ## list of nodes suspected (or added from joins) to proposed as changes by the leader
    self.num_changes = 0     # ## number of nodes in self.changes, i.e., length of self.changes
    self.num_acked = 0       # ## number of nodes in self.changes acknowledged
                               ### set in 1 place by us, using num_changes
    self.num_committed = 0  # ## min of self.num_acked over not suspected nodes
    self.num_installed = 0  # ## number of nodes installed (added/removed) by the node, as proposed by the leader
    self.ragged_edge_computed = False  # ## true for leader calling terminate_epoch or others after leader did;
                                       ## the call happens when leader's num_committed > self's num_installed
    self.active = False  # (p.40) ## true when the epoch is active, only used at start  ### not in pseudocode
                         ### could use logical or over sst[my_rank].suspected or even just own suspected
```

Field "**slots**" is a list of ring buffer with reusable slots for storing client requests and metadata

Specifying system state: view

```
class View():
  """A view that holds the information of an epoch. An epoch is the duration of a view."""
  def __init__(self, n, epoch=0, leader_rank=0):
    # n:  ## number of members in the view
    self.epoch = epoch              # ## epoch number of the view; epoch and view used interchangeably in the paper
    self.leader_rank = leader_rank  # ## index of the leader
    self.members = [None] * n       # ## list of member nodes in the view
    self.failed = [False] * n       # ## for each node, whether that node is suspected and thus considered failed

  def add_member(self, node):         ## add member to the view
    self.members.append(node)         ## append node to members
    self.failed.append(False)         ## add the failed attribute corresponding to the added node

  def remove_member(self, node):      ## remove node from members of the view
    index = self.members.index(node)  ## get index of node in members
    del self.failed[index]            ## remove failed entry for node
    self.members.remove(node)         ## remove node from members
```

View object to store information about a view such as leader and members

Specifying system state: view

```
class View():
  """A view that holds the information of an epoch. An epoch is the duration of a view."""
  def __init__(self, n, epoch=0, leader_rank=0):
    # n:  ## number of members in the view
    self.epoch = epoch              # ## epoch number of the view; epoch and view used interchangeably in the paper
    self.leader_rank = leader_rank  # ## index of the leader
    self.members = [None] * n       # ## list of member nodes in the view
    self.failed = [False] * n       # ## for each node, whether that node is suspected and thus considered failed

  def add_member(self, node):         ## add member to the view
    self.members.append(node)         ## append node to members
    self.failed.append(False)         ## add the failed attribute corresponding to the added node

  def remove_member(self, node):      ## remove node from members of the view
    index = self.members.index(node)  ## get index of node in members
    del self.failed[index]            ## remove failed entry for node
    self.members.remove(node)         ## remove node from members
```

Definition of class **View** with its fields and methods to add or remove members

Specifying view change

Key part of view change algorithm is the leader selection

A leader proposes changes, ie., nodes that needs to be added or removed from the group, during a view change

Specifying view change: leader selection

```
def find_new_leader(r):  # find_new_leader(r) {
  for i in range(len(curr_view.members)):  #   for (int i = 0; i < curr_view.max_rank; ++i) {  ### max_rank replaced
    if sst[r].suspected[i]: continue       #     if (sst[r].suspected[i]) continue;
    else: return i                         #     else return i }}

# (p.35)  ## update the current view, at the end, with the new leader
def leader_selection():                    # always {  ### made function and called in run
  new_leader = find_new_leader(my_rank)    # new_leader = find_new_leader(my_rank)
  if new_leader != curr_view.leader_rank:  # if (new_leader != curr_view.leader_rank && new_leader == my_rank)
    if new_leader == my_rank:              ### split 2 conjuncts, to add the else-branch for the second
    # all_others_agree = True              #   bool all_others_agree = true  ### moved into while-loop
    ### if not moved, if it becomes False in for-loop below, it stays False, and the while-loop never stops
      while find_new_leader(my_rank) == my_rank:  #   while (find_new_leader(my_rank) == my_rank) {
        --receive_messages                 ##    yield to receive msgs
        ### needed to receive updates to SST which may result in new leader selection  ### break atomicity

        all_others_agree = True            ###   moved here from outside while-loop, as explained above
        for r in range(len(sst)):          #     for (r: SST.rows) {
          if not sst[my_rank].suspected[r]:#       if (sst[my_row].suspected[r] == false)
            all_others_agree = all_others_agree and (find_new_leader(r) == my_rank)
                                           #         all_others_agree &&= (find_new_leader(r) == my_rank) }
        if all_others_agree:               #     if (all_others_agree) {
          curr_view.leader_rank = my_rank  #       curr_view.leader_rank = my_rank;
          output("I am the new leader!!!")
          break                            #       break; }}}
    else:                                  ##   else:  ### added else-branch, for when new leader is not self
      curr_view.leader_rank = new_leader   ##     set current view's leader to be new leader
```

Leader selection selects the leader

Figure 6: Specification of leader selection.

Specifying view change: leader selection

```
def find_new_leader(r):  # find_new_leader(r) {
  for i in range(len(curr_view.members)):  #   for (int i = 0; i < curr_view.max_rank; ++i) {  ### max_rank replaced
    if sst[r].suspected[i]: continue       #     if (sst[r].suspected[i]) continue;
    else: return i                         #     else return i }}

# (p.35)  ## update the current view, at the end, with the new leader
def leader_selection():                    # always {  ### made function and called in run
  new_leader = find_new_leader(my_rank)    # new_leader = find_new_leader(my_rank)
  if new_leader != curr_view.leader_rank:  # if (new_leader != curr_view.leader_rank && new_leader == my_rank)
    if new_leader == my_rank:              ### split 2 conjuncts, to add the else-branch for the second
    # all_others_agree = True              #   bool all_others_agree = true  ### moved into while-loop
    ### if not moved, if it becomes False in for-loop below, it stays False, and the while-loop never stops
      while find_new_leader(my_rank) == my_rank:  #   while (find_new_leader(my_rank) == my_rank) {
        --receive_messages                 ##    yield to receive msgs
        ### needed to receive updates to SST which may result in new leader selection  ### break atomicity

        all_others_agree = True            ###   moved here from outside while-loop, as explained above
        for r in range(len(sst)):          #     for (r: SST.rows) {
          if not sst[my_rank].suspected[r]:#       if (sst[my_row].suspected[r] == false)
            all_others_agree = all_others_agree and (find_new_leader(r) == my_rank)
                                           #         all_others_agree &&= (find_new_leader(r) == my_rank) }
        if all_others_agree:               #     if (all_others_agree) {
          curr_view.leader_rank = my_rank  #       curr_view.leader_rank = my_rank;
          output("I am the new leader!!!")
          break                            #       break; }}}
    else:                                  ##   else:  ### added else-branch, for when new leader is not self
      curr_view.leader_rank = new_leader   ##     set current view's leader to be new leader
```

Calls **find_new_leader** to select the new leader

Figure 6: Specification of leader selection.

Specifying view change: leader selection

```
def find_new_leader(r):  # find_new_leader(r) {
  for i in range(len(curr_view.members)):  #   for (int i = 0; i < curr_view.max_rank; ++i) {  ### max_rank replaced
    if sst[r].suspected[i]: continue       #     if (sst[r].suspected[i]) continue;
    else: return i                         #     else return i }}
```

```
# (p.35)  ## update the current view, at the end, with the new leader
def leader_selection():                            # always {  ### made function and called in run
  new_leader = find_new_leader(my_rank)            # new_leader = find_new_leader(my_rank)
  if new_leader != curr_view.leader_rank:          # if (new_leader != curr_view.leader_rank && new_leader == my_rank)
    if new_leader == my_rank:                      ### split 2 conjuncts, to add the else-branch for the second
    # all_others_agree = True                      #   bool all_others_agree = true  ### moved into while-loop
    ### if not moved, if it becomes False in for-loop below, it stays False, and the while-loop never stops
      while find_new_leader(my_rank) == my_rank:   #   while (find_new_leader(my_rank) == my_rank) {
        --receive_messages                         ##    yield to receive msgs
        ### needed to receive updates to SST which may result in new leader selection  ### break atomicity

        all_others_agree = True                    ###   moved here from outside while-loop, as explained above
        for r in range(len(sst)):                  #     for (r: SST.rows) {
          if not sst[my_rank].suspected[r]:        #       if (sst[my_row].suspected[r] == false)
            all_others_agree = all_others_agree and (find_new_leader(r) == my_rank)
                                                   #         all_others_agree &&= (find_new_leader(r) == my_rank) }
        if all_others_agree:                       #     if (all_others_agree) {
          curr_view.leader_rank = my_rank          #       curr_view.leader_rank = my_rank;
          output("I am the new leader!!!")
          break                                    #       break; }}}
    else:                                          ##   else:  ### added else-branch, for when new leader is not self
      curr_view.leader_rank = new_leader           ##     set current view's leader to be new leader
```

Selects the first non-suspected (non-failed) node as the leader.

Figure 6: Specification of leader selection.

Specifying view change: leader selection

```
def find_new_leader(r):  # find_new_leader(r) {
  for i in range(len(curr_view.members)):  #   for (int i = 0; i < curr_view.max_rank; ++i) {  ### max_rank replaced
    if sst[r].suspected[i]: continue       #     if (sst[r].suspected[i]) continue;
    else: return i                         #     else return i }}

# (p.35)  ## update the current view, at the end, with the new leader
def leader_selection():                    # always {  ### made function and called in run
  new_leader = find_new_leader(my_rank)    # new_leader = find_new_leader(my_rank)
  if new_leader != curr_view.leader_rank:  # if (new_leader != curr_view.leader_rank && new_leader == my_rank)
    if new_leader == my_rank:              ### split 2 conjuncts, to add the else-branch for the second
    # all_others_agree = True              #   bool all_others_agree = true  ### moved into while-loop
    ### if not moved, if it becomes False in for-loop below, it stays False, and the while-loop never stops
      while find_new_leader(my_rank) == my_rank:  #   while (find_new_leader(my_rank) == my_rank) {
        --receive_messages                 ##    yield to receive msgs
        ### needed to receive updates to SST which may result in new leader selection  ### break atomicity

        all_others_agree = True            ###   moved here from outside while-loop, as explained above
        for r in range(len(sst)):          #     for (r: SST.rows) {
          if not sst[my_rank].suspected[r]:#       if (sst[my_row].suspected[r] == false)
            all_others_agree = all_others_agree and (find_new_leader(r) == my_rank)
                                           #         all_others_agree &&= (find_new_leader(r) == my_rank) }
        if all_others_agree:               #     if (all_others_agree) {
          curr_view.leader_rank = my_rank  #       curr_view.leader_rank = my_rank;
          output("I am the new leader!!!")
          break                            #       break; }}}
    else:                                  ##   else:  ### added else-branch, for when new leader is not self
      curr_view.leader_rank = new_leader   ##     set current view's leader to be new leader
```

If the new leader is different than the current leader, it waits until all non-suspected nodes recognise it as the leader

Figure 6: Specification of leader selection.

Checking

Manual inspection and automated testing

Systematic runtime checking

Systematic runtime checking

Enabled by general framework for runtime checking in DistAlgo* without changes to the specification

Properties are specified at a high level

Some well-known safety properties we checked are:

- Validity
- Agreement
- Uniform Integrity

* Assurance of Distributed Algorithms and Systems, RV'20

Property checking

Agreement: "If two servers execute the i^{th} update, then these updates are identical"*

each p1.sent('execute', i, req1),
p2.sent('execute', i, req2),
has req1=req2

* Taken from Paxos for System Builders, CNDS'08

Property checking

Agreement: "If two servers execute the i^{th} update, then these updates are identical"*

each p1.sent('execute', i, req1),
p2.sent('execute', i, req2),
has req1=req2

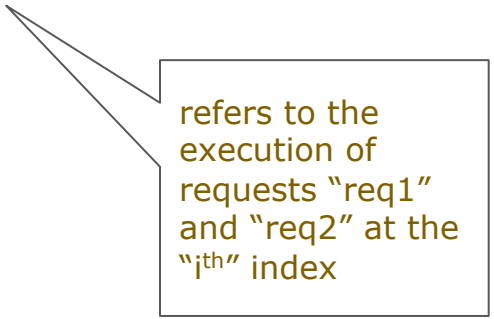
p1, p2 are processes (used in place of servers)

* Taken from Paxos for System Builders, CNDS'08

Property checking

Agreement: "If two servers execute the i^{th} update, then these updates are identical"*

each p1.sent('execute', i, req1),
p2.sent('execute', i, req2),
has req1=req2



* Taken from Paxos for System Builders, CNDS'08

Property checking

Agreement: "If two servers execute the i^{th} update, then these updates are identical"*

each p1.sent('execute', i, req1),
p2.sent('execute', i, req2),
has req1=req2

for each, p1 that executes req1 at the i^{th} index

* Taken from Paxos for System Builders, CNDS'08

Property checking

Agreement: "If two servers execute the i^{th} update, then these updates are identical"*

each p1.sent('execute', i, req1),
p2.sent('execute', i, req2),
has req1=req2

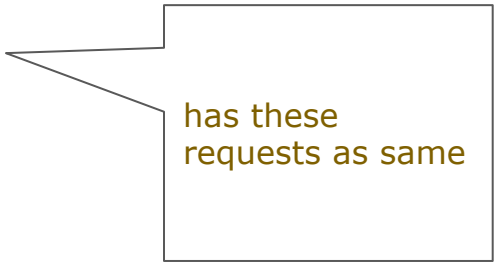
and p2 that executes req2 at i^{th} index

* Taken from Paxos for System Builders, CNDS'08

Property checking

Agreement: "If two servers execute the i^{th} update, then these updates are identical"*

each p1.sent('execute', i, req1),
p2.sent('execute', i, req2),
has req1=req2

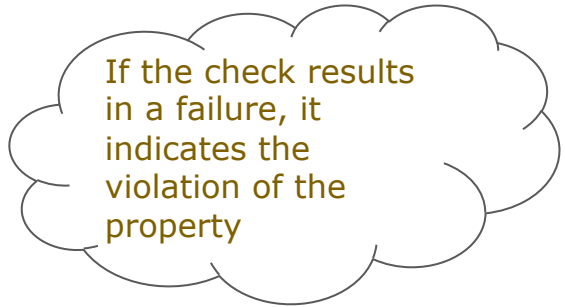


* Taken from Paxos for System Builders, CNDS'08

Property checking

Agreement: "If two servers execute the i^{th} update, then these updates are identical"*

each p1.sent('execute', i, req1),
p2.sent('execute', i, req2),
has req1=req2

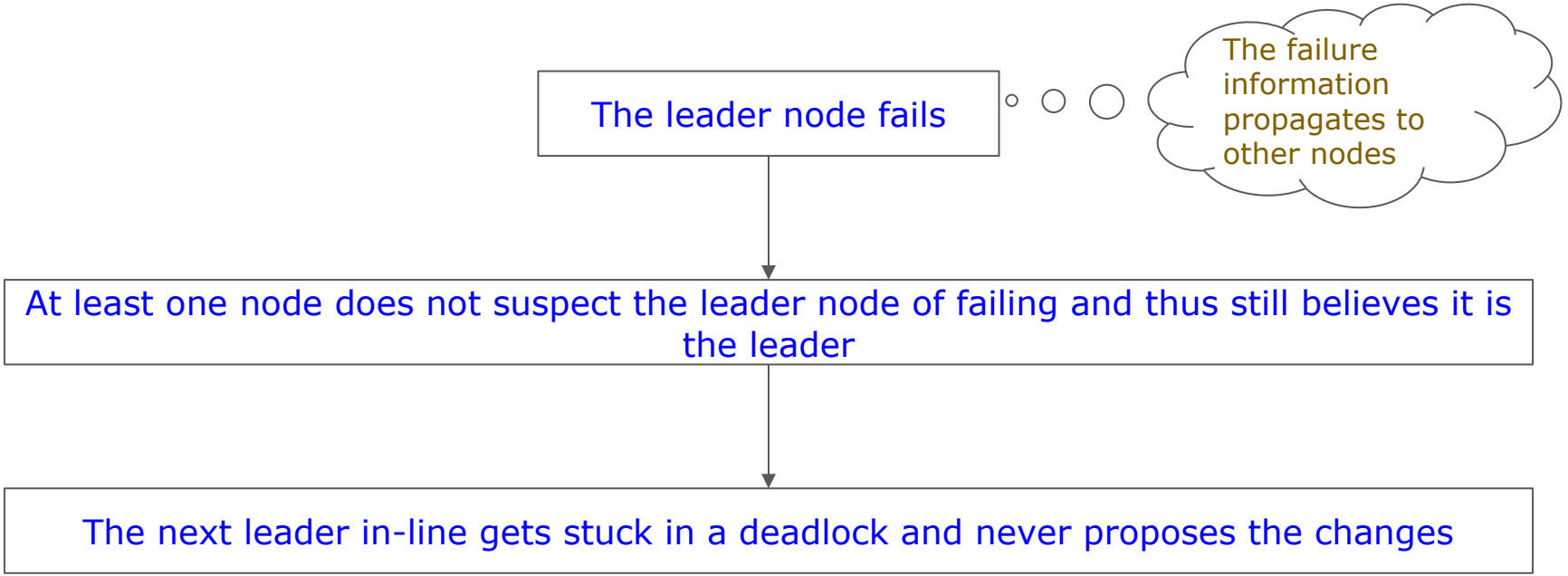


* Taken from Paxos for System Builders, CNDS'08

Issues found and fixed

- Bugs in pseudocode are quite normal, since pseudocode is manually created and there is no way to run or check
- Specification and checking approach helped identify some issues in Derecho pseudocode
- Two examples presented in paper
  - Overwriting in ring buffer
  - Deadlock in leader selection

Deadlock in leader selection



Deadlock in leader selection

```
def find_new_leader(r):  # find_new_leader(r) {
  for i in range(len(curr_view.members)):  #   for (int i = 0; i < curr_view.max_rank; ++i) {  ### max_rank replaced
    if sst[r].suspected[i]: continue       #     if (sst[r].suspected[i]) continue;
    else: return i                         #     else return i }}

# (p.35)  ## update the current view, at the end, with the new leader
def leader_selection():                    # always {  ### made function and called in run
  new_leader = find_new_leader(my_rank)    # new_leader = find_new_leader(my_rank)
  if new_leader != curr_view.leader_rank:  # if (new_leader != curr_view.leader_rank && new_leader == my_rank)
    if new_leader == my_rank:              ### split 2 conjuncts, to add the else-branch for the second
    # all_others_agree = True              #   bool all_others_agree = true  ### moved into while-loop
    ### if not moved, if it becomes False in for-loop below, it stays False, and the while-loop never stops
      while find_new_leader(my_rank) == my_rank:  #   while (find_new_leader(my_rank) == my_rank) {
        --receive_messages                ##    yield to receive msgs
        ### needed to receive updates to SST which may result in new leader selection  ### break atomicity

        all_others_agree = True           ###   moved here from outside while-loop, as explained above
        for r in range(len(sst)):         #     for (r: SST.rows) {
          if not sst[my_rank].suspected[r]:  #       if (sst[my_row].suspected[r] == false)
            all_others_agree = all_others_agree and (find_new_leader(r) == my_rank)
                                          #         all_others_agree &&= (find_new_leader(r) == my_rank) }
        if all_others_agree:              #     if (all_others_agree) {
          curr_view.leader_rank = my_rank #       curr_view.leader_rank = my_rank;
          output("I am the new leader!!!")
          break                           #       break; }}}
    else:                                 ##   else:  ### added else-branch, for when new leader is not self
      curr_view.leader_rank = new_leader  ##     set current view's leader to be new leader
```

If the new leader is different than the current leader, it waits until all non-suspected nodes recognise it as the leader

Figure 6: Specification of leader selection.

Deadlock in leader selection

```
def find_new_leader(r):  # find_new_leader(r) {
  for i in range(len(curr_view.members)):  #   for (int i = 0; i < curr_view.max_rank; ++i) {  ### max_rank replaced
    if sst[r].suspected[i]: continue       #     if (sst[r].suspected[i]) continue;
    else: return i                         #     else return i }}

# (p.35)  ## update the current view, at the end, with the new leader
def leader_selection():                    # always {  ### made function and called in run
  new_leader = find_new_leader(my_rank)    # new_leader = find_new_leader(my_rank)
  if new_leader != curr_view.leader_rank:  # if (new_leader != curr_view.leader_rank && new_leader == my_rank)
    if new_leader == my_rank:              ### split 2 conjuncts, to add the else-branch for the second
    # all_others_agree = True              #   bool all_others_agree = true  ### moved into while-loop
    ### if not moved, if it becomes False in for-loop below, it stays False, and the while-loop never stops
      while find_new_leader(my_rank) == my_rank:  #   while (find_new_leader(my_rank) == my_rank) {
        --receive_messages                 ##    yield to receive msgs
        ### needed to receive updates to SST which may result in new leader selection  ### break atomicity

        all_others_agree = True            ###   moved here from outside while-loop, as explained above
        for r in range(len(sst)):          #     for (r: SST.rows) {
          if not sst[my_rank].suspected[r]:  #       if (sst[my_row].suspected[r] == false)
            all_others_agree = all_others_agree and (find_new_leader(r) == my_rank)
                                           #         all_others_agree &&= (find_new_leader(r) == my_rank) }
        if all_others_agree:               #     if (all_others_agree) {
          curr_view.leader_rank = my_rank  #       curr_view.leader_rank = my_rank;
          output("I am the new leader!!!")
          break                            #       break; }}}
    else:                                  ##   else:  ### added else-branch, for when new leader is not self
      curr_view.leader_rank = new_leader   ##     set current view's leader to be new leader
```

Uses logical conjunction to check if all the nodes agree with the new leader selection

Figure 6: Specification of leader selection.

Deadlock in leader selection

```
def find_new_leader(r):  # find_new_leader(r) {
  for i in range(len(curr_view.members)):  #   for (int i = 0; i < curr_view.max_rank; ++i) {  ### max_rank replaced
    if sst[r].suspected[i]: continue       #     if (sst[r].suspected[i]) continue;
    else: return i                         #     else return i }}

# (p.35)  ## update the current view, at the end, with the new leader
def leader_selection():                    # always {  ### made function and called in run
  new_leader = find_new_leader(my_rank)    # new_leader = find_new_leader(my_rank)
  if new_leader != curr_view.leader_rank:  # if (new_leader != curr_view.leader_rank && new_leader == my_rank)
    if new_leader == my_rank:              ### split 2 conjuncts, to add the else-branch for the second
    # all_others_agree = True              #   bool all_others_agree = true  ### moved into while-loop
    ### if not moved, if it becomes False in for-loop below, it stays False, and the while-loop never stops
      while find_new_leader(my_rank) == my_rank:  #   while (find_new_leader(my_rank) == my_rank) {
        --receive_messages                 ##    yield to receive msgs
        ### needed to receive updates to SST which may result in new leader selection  ### break atomicity

        all_others_agree = True            ###   moved here from outside while-loop, as explained above
        for r in range(len(sst)):          #     for (r: SST.rows) {
          if not sst[my_rank].suspected[r]:#       if (sst[my_row].suspected[r] == false)
            all_others_agree = all_others_agree and (find_new_leader(r) == my_rank)
                                           #         all_others_agree &&= (find_new_leader(r) == my_rank) }
        if all_others_agree:               #     if (all_others_agree) {
          curr_view.leader_rank = my_rank  #       curr_view.leader_rank = my_rank;
          output("I am the new leader!!!")
          break                            #       break; }}}
    else:                                  ##   else:  ### added else-branch, for when new leader is not self
      curr_view.leader_rank = new_leader   ##     set current view's leader to be new leader
```

Consider that if a node does not initially agree, ie, **find_new_leader** returned different leader

Figure 6: Specification of leader selection.

Deadlock in leader selection

```
def find_new_leader(r):  # find_new_leader(r) {
  for i in range(len(curr_view.members)):  #   for (int i = 0; i < curr_view.max_rank; ++i) {  ### max_rank replaced
    if sst[r].suspected[i]: continue       #     if (sst[r].suspected[i]) continue;
    else: return i                         #     else return i }}

# (p.35)  ## update the current view, at the end, with the new leader
def leader_selection():                    # always {  ### made function and called in run
  new_leader = find_new_leader(my_rank)    # new_leader = find_new_leader(my_rank)
  if new_leader != curr_view.leader_rank:  # if (new_leader != curr_view.leader_rank && new_leader == my_rank)
    if new_leader == my_rank:              ### split 2 conjuncts, to add the else-branch for the second
    # all_others_agree = True              #   bool all_others_agree = true  ### moved into while-loop
    ### if not moved, if it becomes False in for-loop below, it stays False, and the while-loop never stops
      while find_new_leader(my_rank) == my_rank:  #   while (find_new_leader(my_rank) == my_rank) {
        --receive_messages                 ##    yield to receive msgs
        ### needed to receive updates to SST which may result in new leader selection  ### break atomicity

        all_others_agree = True            ###   moved here from outside while-loop, as explained above
        for r in range(len(sst)):          #     for (r: SST.rows) {
          if not sst[my_rank].suspected[r]:  #       if (sst[my_row].suspected[r] == false)
            all_others_agree = all_others_agree and (find_new_leader(r) == my_rank)
                                           #         all_others_agree &&= (find_new_leader(r) == my_rank) }
        if all_others_agree:               #     if (all_others_agree) {
          curr_view.leader_rank = my_rank  #       curr_view.leader_rank = my_rank;
          output("I am the new leader!!!")
          break                            #       break; }}}
    else:                                  ##   else:  ### added else-branch, for when new leader is not self
      curr_view.leader_rank = new_leader   ##     set current view's leader to be new leader
```

all_others_agree would be set to **false** and never be set to true again

Figure 6: Specification of leader selection.

Deadlock in leader selection

```
def find_new_leader(r):  # find_new_leader(r) {
  for i in range(len(curr_view.members)):  #   for (int i = 0; i < curr_view.max_rank; ++i) {  ### max_rank replaced
    if sst[r].suspected[i]: continue       #     if (sst[r].suspected[i]) continue;
    else: return i                         #     else return i }}

# (p.35)  ## update the current view, at the end, with the new leader
def leader_selection():                    # always {  ### made function and called in run
  new_leader = find_new_leader(my_rank)    # new_leader = find_new_leader(my_rank)
  if new_leader != curr_view.leader_rank:  # if (new_leader != curr_view.leader_rank && new_leader == my_rank)
    if new_leader == my_rank:              ### split 2 conjuncts, to add the else-branch for the second
    # all_others_agree = True              #   bool all_others_agree = true  ### moved into while-loop
    ### if not moved, if it becomes False in for-loop below, it stays False, and the while-loop never stops
      while find_new_leader(my_rank) == my_rank:  #   while (find_new_leader(my_rank) == my_rank) {
        --receive_messages                 ##    yield to receive msgs
        ### needed to receive updates to SST which may result in new leader selection  ### break atomicity

        all_others_agree = True            ###   moved here from outside while-loop, as explained above
        for r in range(len(sst)):          #     for (r: SST.rows) {
          if not sst[my_rank].suspected[r]:#       if (sst[my_row].suspected[r] == false)
            all_others_agree = all_others_agree and (find_new_leader(r) == my_rank)
                                           #         all_others_agree &&= (find_new_leader(r) == my_rank) }
        if all_others_agree:               #     if (all_others_agree) {
          curr_view.leader_rank = my_rank  #       curr_view.leader_rank = my_rank;
          output("I am the new leader!!!")
          break                            #       break; }}}
    else:                                  ##   else:  ### added else-branch, for when new leader is not self
      curr_view.leader_rank = new_leader   ##     set current view's leader to be new leader
```

and the leader selection would never succeed

Figure 6: Specification of leader selection.

Deadlock in leader selection

```
def find_new_leader(r):  # find_new_leader(r) {
  for i in range(len(curr_view.members)):  #   for (int i = 0; i < curr_view.max_rank; ++i) {  ### max_rank replaced
    if sst[r].suspected[i]: continue       #     if (sst[r].suspected[i]) continue;
    else: return i                         #     else return i }}

# (p.35)  ## update the current view, at the end, with the new leader
def leader_selection():                    # always {  ### made function and called in run
  new_leader = find_new_leader(my_rank)    # new_leader = find_new_leader(my_rank)
  if new_leader != curr_view.leader_rank:  # if (new_leader != curr_view.leader_rank && new_leader == my_rank)
    if new_leader == my_rank:              ### split 2 conjuncts, to add the else-branch for the second
    # all_others_agree = True              #   bool all_others_agree = true  ### moved into while-loop
    ### if not moved, if it becomes False in for-loop below, it stays False, and the while-loop never stops
      while find_new_leader(my_rank) == my_rank:  #   while (find_new_leader(my_rank) == my_rank) {
        --receive_messages                 ##    yield to receive msgs
        ### needed to receive updates to SST which may result in new leader selection  ### break atomicity

        all_others_agree = True            ###   moved here from outside while-loop, as explained above
        for r in range(len(sst)):          #     for (r: SST.rows) {
          if not sst[my_rank].suspected[r]:#       if (sst[my_row].suspected[r] == false)
            all_others_agree = all_others_agree and (find_new_leader(r) == my_rank)
                                           #         all_others_agree &&= (find_new_leader(r) == my_rank) }
        if all_others_agree:               #     if (all_others_agree) {
          curr_view.leader_rank = my_rank  #       curr_view.leader_rank = my_rank;
          output("I am the new leader!!!")
          break                            #       break; }}}
    else:                                  ##   else:  ### added else-branch, for when new leader is not self
      curr_view.leader_rank = new_leader   ##     set current view's leader to be new leader
```

to resolve this, we moved the initialization to here, inside the while loop

Figure 6: Specification of leader selection.

Deadlock in leader selection

```
def find_new_leader(r):  # find_new_leader(r) {
  for i in range(len(curr_view.members)):  #   for (int i = 0; i < curr_view.max_rank; ++i) {  ### max_rank replaced
    if sst[r].suspected[i]: continue       #     if (sst[r].suspected[i]) continue;
    else: return i                         #     else return i }}

# (p.35)  ## update the current view, at the end, with the new leader
def leader_selection():                    # always {  ### made function and called in run
  new_leader = find_new_leader(my_rank)    # new_leader = find_new_leader(my_rank)
  if new_leader != curr_view.leader_rank:  # if (new_leader != curr_view.leader_rank && new_leader == my_rank)
    if new_leader == my_rank:              ### split 2 conjuncts, to add the else-branch for the second
    # all_others_agree = True              #   bool all_others_agree = true  ### moved into while-loop
    ### if not moved, if it becomes False in for-loop below, it stays False, and the while-loop never stops
      while find_new_leader(my_rank) == my_rank:  #   while (find_new_leader(my_rank) == my_rank) {
        --receive_messages                 ##    yield to receive msgs
        ### needed to receive updates to SST which may result in new leader selection  ### break atomicity

        all_others_agree = True            ###   moved here from outside while-loop, as explained above
        for r in range(len(sst)):          #     for (r: SST.rows) {
          if not sst[my_rank].suspected[r]:#       if (sst[my_row].suspected[r] == false)
            all_others_agree = all_others_agree and (find_new_leader(r) == my_rank)
                                           #         all_others_agree &&= (find_new_leader(r) == my_rank) }
        if all_others_agree:               #     if (all_others_agree) {
          curr_view.leader_rank = my_rank  #       curr_view.leader_rank = my_rank;
          output("I am the new leader!!!")
          break                            #       break; }}}
    else:                                  ##   else:  ### added else-branch, for when new leader is not self
      curr_view.leader_rank = new_leader   ##     set current view's leader to be new leader
```

and added an else branch to update the new leader information for non-leaders

Figure 6: Specification of leader selection.

Resulting specification size

Table 1: Specification size (in number of lines, including output lines, excluding empty or comment-only lines) for Derecho specification in DistAlgo (`derecho.da` in [40, Appendix A] excluding method `main` and class `Sim`).

Protocol component	Size
state and helper functions	95
steady-state execution, incl. delivering&executing reqs	63
view change	132
`imports`, helper, choices in `run`	14
total	304

Conclusion

- Precise, directly executable specification of Derecho
- Runtime checking of important safety properties of Derecho

Future work

- Implementing more fault injection in checking the protocol
- Use of DistAlgo specification to help with the proof development by the Derecho team for both safety and liveness
- Automated ways to correlate formal specification in DistAlgo with implementations in lower-level languages such as C++

Thank You