From Clarity to Efficiency for Distributed Algorithms

YANHONG A. LIU, Stony Brook University
SCOTT D. STOLLER, Stony Brook University
BO LIN, Stony Brook University

This article describes a very high-level language for clear description of distributed algorithms and optimizations necessary for generating efficient implementations. The language supports high-level control flows where complex synchronization conditions can be expressed using high-level queries, especially logic quantifications, over message history sequences. Unfortunately, the programs would be extremely inefficient, including consuming unbounded memory, if executed straightforwardly.

We present new optimizations that automatically transform complex synchronization conditions into incremental updates of necessary auxiliary values as messages are sent and received. The core of the optimizations is the first general method for efficient implementation of logic quantifications. We have developed an operational semantics of the language, implemented a prototype of the compiler and the optimizations, and successfully used the language and implementation on a variety of important distributed algorithms.

CCS Concepts: • **Information systems** → **Query optimization**; • **Theory of computation** → **Logic**; *Operational semantics*; • **Software and its engineering** → **Distributed programming languages**; **Very high level languages**; **Concurrent programming structures**; *Compilers*;

Additional Key Words and Phrases: distributed algorithms, high-level queries and updates, incrementalization, logic quantifications, message histories, synchronization conditions, yield points

ACM Reference format:
Yanhong A. Liu, Scott D. Stoller, and Bo Lin. 2017. From Clarity to Efficiency for Distributed Algorithms . *ACM Trans. Program. Lang. Syst.* 1, 1, Article 1 (January 2017), 40 pages.
DOI: 0000001.0000001

1 INTRODUCTION

Distributed algorithms are at the core of distributed systems. Yet, developing practical implementations of distributed algorithms with correctness and efficiency assurances remains a challenging, recurring task.

- Study of distributed algorithms has relied on either pseudocode with English, which is high-level but imprecise, or formal specification languages, which are precise but harder to understand, lacking mechanisms for building real distributed systems, or not executable at all.

This work was supported in part by NSF under grants CCF-1414078, CCF-1248184, CCF-0964196, CNS-0831298, and CCF-0613913; and ONR under grants N000141512208, N000140910651 and N000140710928.
Author's address: Computer Science Department, Stony Brook University, Stony Brook, NY 11794, USA. {liu,stoller,bolin}@cs.stonybrook.edu .

DOI: 0000001.0000001

- At the same time, programming of distributed systems has mainly been concerned with program efficiency and has relied mostly on the use of low-level or complex libraries and to a lesser extent on built-in mechanisms in restricted programming models.

What's lacking is (1) a simple and powerful language that can express distributed algorithms at a high level and yet has a clear semantics for precise execution as well as for verification, and is fully integrated into widely used programming languages for building real distributed systems, together with (2) powerful optimizations that can transform high-level algorithm descriptions into efficient implementations.

This article describes a very high-level language, DistAlgo, for clear description of distributed algorithms, combining advantages of pseudocode, formal specification languages, and programming languages.

- The main control flow of a process, including sending messages and waiting on conditions about received messages, can be stated directly as in sequential programs; yield points where message handlers execute can be specified explicitly and declaratively.
- Complex synchronization conditions can be expressed using high-level queries, especially quantifications, over message history sequences, without manually writing message handlers that perform low-level incremental updates and obscure control flows.

DistAlgo supports these features by building on an object-oriented programming language. We also developed an operational semantics for the language. The result is that distributed algorithms can be expressed in DistAlgo clearly at a high level, like in pseudocode, but also precisely, like in formal specification languages, facilitating formal verification, and can be executed as part of real applications, as in programming languages.

Unfortunately, programs containing control flows with synchronization conditions expressed at such a high level are extremely inefficient if executed straightforwardly: each quantifier can introduce a linear factor in running time, and any use of the history of messages sent and received may cause space usage to be unbounded.

We present new optimizations that allow efficient implementations to be generated automatically, extending previous optimizations to distributed programs and to the most challenging quantifications.

- Our method transforms sending and receiving of messages into updates to message history sequences, incrementally maintains the truth values of synchronization conditions and necessary auxiliary values as those sequences are updated, and finally removes those sequences as dead code when appropriate.
- To incrementally maintain the truth values of general quantifications, our method first transforms them into aggregations, also called aggregate queries. In general, however, translating nested quantifications simply into nested aggregations can incur asymptotically more space and time overhead than necessary. Our transformations minimize the nesting of the resulting queries.
- Quantified order comparisons are used extensively in nontrivial distributed algorithms. They can be incrementalized easily when not mixed with other conditions or with each other. We systematically extract single quantified order comparisons and transform them into efficient incremental operations.

Overall, our method significantly improves time complexities and reduces the unbounded space used for message history sequences to the auxiliary space needed for incremental computation. Systematic incrementalization also allows the time and space complexity of the generated programs to be analyzed easily.

There has been a significant amount of related research, as discussed in Section 7. Our work contains three main contributions:

- A simple and powerful language for expressing distributed algorithms with high-level control flows and synchronization conditions, an operational semantics, and full integration into an object-oriented language.
- A systematic method for incrementalizing complex synchronization conditions with respect to all sending and receiving of messages in distributed programs.
- A general and systematic method for generating efficient implementations of arbitrary logic quantifications together with general high-level queries.

We have implemented a prototype of the compiler and the optimizations and experimented with a variety of important distributed algorithms, including Paxos, Byzantine Paxos, and multi-Paxos. Our experiments strongly confirm the benefits of the language and the effectiveness of the optimizations.

This article is a revised version of [56]. The main changes are revised and extended descriptions of the language and the optimization method, a new formal operational semantics, an abridged and updated description of the implementation, and a new description of our experience of using DistAlgo in teaching.

2 EXPRESSING DISTRIBUTED ALGORITHMS

Even when a distributed algorithm appears simple at a high level, it can be subtle when necessary details are considered, making it difficult to understand how the algorithm works precisely. The difficulty comes from the fact that multiple processes must coordinate and synchronize to achieve global goals, but at the same time, delays, failures, and attacks can occur. Even determining the ordering of events is nontrivial, which is why Lamport's logical clock [43] is so fundamental for distributed systems.

Running example. We use Lamport's distributed mutual exclusion algorithm [43] as a running example. Lamport developed it to illustrate the logical clock he invented. 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 contains Lamport's original description of the algorithm, except with the notation $<$ instead of $\longrightarrow$ in rule 5 (for comparing pairs of timestamps and process ids using lexical ordering: `(a,b)` $<$ `(a2,b2)` iff `a` $<$ `a2` or `a` $=$ `a2` and `b` $<$ `b2`) and with the word "acknowledgment" added in rule 5 (for simplicity when omitting a commonly omitted [29, 59] 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 satisfies safety, liveness, and fairness, and has a message complexity of $3(n-1)$. It 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 timestamps of the request messages. Its message complexity is $3(n-1)$ in that $3(n-1)$ messages are required to serve each request.

Challenges. To understand how this algorithm is carried out precisely, one must understand how each of the n processes acts as both P_i and P_j in interactions with all other processes. Each process must have an order of handling all the events according to the five rules, trying to reach its own goal

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.

Fig. 1. Original description in English.

of entering and exiting a critical section while also responding to messages from other processes. It must also keep testing the complex condition in rule 5 as events happen.

State machine based formal specifications have been used to fill in such details precisely, but at the same time, they are lower-level and harder to understand. For example, a formal specification of Lamport's algorithm in I/O automata [59, pages 647-648] occupies about one and a fifth pages, most of which is double-column.

To actually implement distributed algorithms, details for many additional aspects must be added, for example, creating processes, letting them establish communication channels with each other, incorporating appropriate logical clocks (e.g., Lamport clock or vector clock [60]) if needed, guaranteeing the specified channel properties (e.g., reliable, FIFO), and integrating the algorithm with the application (e.g., specifying critical section tasks and invoking the code for the algorithm as part of the overall application). Furthermore, how to do all of these in an easy and modular fashion?

Our approach. We address these challenges with the DistAlgo language, compilation to executable programs, and especially optimization by incrementalization of expensive synchronizations, described in Sections 3, 4, and 5, respectively. An unexpected result is that incrementalization led us to discover simplifications of Lamport's original algorithm in Figure 1; the simplified algorithm can be expressed using basically two `send` statements, a `receive` definition, and an `await` statement.

The results on the running example are shown in Figures 2–5, with details explained later. Figure 2 shows Lamport's original algorithm expressed in DistAlgo; it also includes configuration and setup for running 50 processes each trying to enter critical section at some point during its execution. Figures 3 and 4 show two alternative optimized programs after incrementalization; all lines with comments are new except that the `await` statement is simplified. Figure 5 shows the simplified algorithm.

3 DISTALGO LANGUAGE

To support distributed programming at a high level, four main concepts can be added to commonly used high-level programming languages, especially object-oriented languages, such as Python and Java: (1) distributed processes, and sending messages, (2) control flows with yield points and waits, and receiving messages, (3) synchronization conditions using high-level queries of message history sequences, and (4) configuration of processes and communication mechanisms. DistAlgo supports these concepts, with options and generalizations for ease of programming, as described below. A formal operational semantics for DistAlgo is presented in Appendix A.

Processes and sending of messages. Distributed processes are concurrent executions of programmed instructions, like threads in Java and Python, except that each process has its private memory, not shared with other processes, and processes communicate by message passing. Three main constructs are used, for defining processes, creating processes, and sending messages.

A process definition is of form (1) below. It defines a type p of processes, by defining a class p that extends class `process`. The *process_body* is a set of method definitions and handler definitions, to be described.

```
class p extends process:
  process_body
```
(1)

A special method `setup` may be defined in *process_body* for initially setting up data in the process before the process's execution starts. A special method `run()` may be defined in *process_body* for carrying out the main flow of execution. A special variable `self` refers to the process itself.

A process creation statement is of form (2) below. It creates n new processes of type p at each node in the value of expression *node_exp*, and returns the resulting process or set of processes. A node is a running DistAlgo program on a machine, and is identified by the host name of the machine plus the name of the running DistAlgo program that can be specified when starting the program.

```
n new p at node_exp
```
(2)

The number n and the `at` clause are optional; the defaults are 1 and the local node, respectively. A new process can be set up by calling its `setup` method. A call `start()` on the process then starts the execution of its `run()` method.

A statement for sending messages is of form (3) below. It sends the message that is the value of expression *mexp* to the process or set of processes that is the value of expression *pexp*.

```
send mexp to pexp
```
(3)

A message can be any value but is by convention a tuple whose first component is a string, called a tag, indicating 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. Three main constructs are used, for specifying yield points, handling of received messages, and synchronization.

A yield point preceding a statement is of form (4) below, where identifier l is a label. It specifies that point in the program as a place where control yields to handling of un-handled messages, if any, and resumes afterwards.

```
-- l
```
(4)

The label `l` is optional; it can be omitted when this yield point is not explicitly referred to in any handler definitions, defined next.

A handler definition, also called a `receive` definition, is of form (5) below. It handles, at yield points labeled l_1, ..., l_j, un-handled messages that match some $mexp_i$ sent from $pexp_i$, where $mexp_i$ and $pexp_i$ are parts of a tuple pattern; previously unbound variables in a pattern are bound to the corresponding components in the value matched. The *handler_body* is a sequence of statements to be executed for the matched messages.

```
receive mexp_1 from pexp_1, ..., mexp_k from pexp_k
    at l_1, ..., l_j:                                    (5)
  handler_body
```

The `from` and `at` clauses are optional; the defaults are any process and all yield points, respectively. If the `from` clause is used, each message is automatically extended with the process id of the sender. A tuple pattern is a tuple in which each component is a non-variable expression, a variable possibly prefixed with "=", a wildcard, or recursively a tuple pattern. A non-variable expression or a variable prefixed with "=" means that the corresponding component of the tuple being matched must equal the value of the non-variable expression or the variable, respectively, for pattern matching to succeed. A variable not prefixed with "=" matches any value and becomes bound to the corresponding component of the tuple being matched. A wildcard, written as "_", matches any value. Support for `receive` mimics common usage in pseudocode, allowing a message handler to be associated with multiple yield points without using method definition and invocations. As syntactic sugar, a `receive` that is handled at only one yield point can be written at that point.

Synchronization and associated actions can be expressed using general, nondeterministic `await` statements. A simple `await` statement is one of the two forms in (6) below. It waits for the value of Boolean-valued expression *bexp* to become true, for the first form, or waits for a timeout after time period t, for the second form.

```
await bexp
await timeout t                                          (6)
```

A general, nondeterministic `await` statement is of form (7) below. It waits for any of the values of expressions $bexp_1$, ..., $bexp_k$ to become true or a timeout after time period t, and then nondeterministically selects one of statements $stmt_1$, ..., $stmt_k$, *stmt* whose corresponding conditions are satisfied to execute. The `or` and `timeout` clauses are optional.

```
await bexp_1: stmt_1
or ...
or bexp_k: stmt_k                                        (7)
timeout t: stmt
```

An `await` statement must be preceded by a yield point, for handling messages while waiting; if a yield point is not specified explicitly, the default is that all message handlers can be executed at this point.

These few constructs make it easy to specify any process that has its own flow of control while also responding to messages. It is also easy to specify any process that only responds to messages, for example, by writing just `receive` definitions and a `run()` method containing only `await false`.

Synchronization conditions using high-level queries. Synchronization conditions and other conditions can be expressed using high-level queries—quantifications, comprehensions, and aggregations—over sets of processes and sequences of messages. High-level queries are used commonly in distributed algorithms because (1) they make complex synchronization conditions clearer and easier to write, and (2) the complexity of distributed algorithms is measured by round complexity and message complexity, not time complexity of local processing.

Quantifications are especially common because they directly capture the truth values of synchronization conditions. We discovered a number of errors in our initial programs that were written using aggregations in place of quantifications before we developed the method to systematically optimize quantifications. For example, we regularly expressed "v is larger than all elements of s" as `v > max s` and either forgot to handle the case that `s` is empty or handled it in an ad hoc fashion. Naive use of aggregation operators like `max` may also hinder generation of more efficient implementations.

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.

- A quantification is a query of one of the two forms in (8) below, called existential and universal quantifications, respectively, plus a set of parameters—variables whose values are bound before the query. For a query to be well-formed, every variable in it must be reachable from a parameter—be a parameter or recursively be the left-side variable of a membership clause whose right-side variables are reachable. Given values of parameters, the query returns `true` iff for some or all, respectively, combinations of values of variables that satisfy all membership clauses v_i `in` $sexp_i$, expression $bexp$ evaluates to `true`. When an existential quantification returns `true`, all variables in the query are also bound to a combination of values, called a witness, that satisfy all the membership clauses and condition $bexp$.

  $$\begin{array}{l}\texttt{some } v_1 \texttt{ in } sexp_1\texttt{, ..., } v_k \texttt{ in } sexp_k \texttt{ | } bexp \\ \texttt{each } v_1 \texttt{ in } sexp_1\texttt{, ..., } v_k \texttt{ in } sexp_k \texttt{ | } bexp\end{array} \tag{8}$$

  For example, the following query returns `true` iff each element in `s` is greater than each element in `s2`.

  ```
  each x in s, x2 in s2 | x > x2
  ```

  For another example, the following query, containing a nested quantification, returns `true` iff some element in `s` is greater than each element in `s2`. Additionally, when the query returns true, variable `x` is bound to a witness—an element in `s` that is greater than each element in `s2`.

  ```
  some x in s | each x2 in s2 | x > x2
  ```

- A comprehension is a query of form (9) below. Given values of parameters, the query returns the set of values of exp for all combinations of values of variables that satisfy all membership clauses v_i `in` $sexp_i$ and condition $bexp$.

  $$\texttt{\{ } exp\texttt{: } v_1 \texttt{ in } sexp_1\texttt{, ..., } v_k \texttt{ in } sexp_k \texttt{| } bexp \texttt{ \}} \tag{9}$$

  For example, the following query returns the set of products of `x` in `s` and `x2` in `s2` where `x` is greater than `x2`.

  ```
  {x*y: x in s, x2 in s2 | x > x2}
  ```

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

- An aggregation, also called an aggregate query, is a query of one of the two forms in (10) below, where agg is an aggregation operator, including `count`, `sum`, `min`, and `max`. Given values of parameters, the query returns the value of applying agg to the set value of $sexp$, for the first form, or to the multiset of values of exp for all combinations of values of variables that satisfy all membership clauses v_i `in` $sexp_i$ and condition $bexp$, for the second form.

  $$\begin{array}{l}agg\ sexp \\ agg\ \texttt{\{ } exp\texttt{: } v_1 \texttt{ in } sexp_1\texttt{, ..., } v_k \texttt{ in } sexp_k \texttt{| } bexp \texttt{ \}}\end{array} \tag{10}$$

- In the query forms above, each v_i can also be a tuple pattern t_i. Variables in t_i are bound to the corresponding components in the matched elements of the value of $sexp_i$. We omit `| bexp` when *bexp* is `true`.

We use `{}` for empty set; use `s.add(x)` and `s.del(x)` for element addition and deletion, respectively; and use `x in s` and `x not in s` for membership test and its negation, respectively. We assume that hashing is used in implementing sets, and the expected time of set initialization, element addition and removal, and membership test is $O(1)$. We consider operations that involve iterations over sets and sequences to be expensive; each iteration over a set or sequence incurs a cost that is linear in the size of the set or sequence. All quantifications, comprehensions, and aggregations are considered expensive.

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

- Sequence `received` is updated only at yield points; after a message arrives, it will be handled when execution reaches the next yield point, by adding the message to `received` and running matching `receive` definitions, if any, associated with the yield point. We use `received m from p` interchangeably with `m from p in received` to mean that message `m` from process `p` is in `received`; `from p` is optional, but when specified, each message in `received` is automatically extended with the process id of the sender.
- Sequence `sent` is updated at each `send` statement; each message sent to a process is added to `sent`. We use `sent m to p` interchangeably with `m to p in sent` to mean that message `m` to process `p` is in `sent`; `to p` is optional, but when specified, `p` is the process to which `m` was sent as specified in the `send` statement.

If implemented straightforwardly, `received` and `sent` can create a huge memory leak, because they can grow unboundedly, preventing their use in practical programming. Our method can remove them by maintaining only auxiliary values that are needed for incremental computation.

Configuration. One can specify channel types, handling of messages, and other configuration items. Such specifications are declarative, so that algorithms can be expressed without unnecessary implementation details. We describe a few basic kinds of configuration items.

First, one can specify the types of channels for passing messages. For example, the following statement configures all channels to be FIFO.

```
configure channel = fifo
```

Other options for `channel` include `reliable` and `{reliable, fifo}`. When either `fifo` or `reliable` is included, TCP is used for process communication; otherwise, UDP is used. In general, channels can also be configured separately for messages from any set of processes to any set of processes.

One can specify how much effort is spent processing messages at yield points. For example,

```
configure handling = all
```

configures the system to handle all un-handled messages at each yield point; this is the default. For another example, one can specify a time limit. One can also specify different handling effort for different yield points.

Logical clocks [27, 43, 60] are used in many distributed algorithms. One can specify the logical clock, e.g., Lamport clock, that is used:

```
configure clock = Lamport
```

It configures sending and receiving of messages to update the clock appropriately. A call `logical_time()` returns the current value of the logical clock.

Overall, a DistAlgo program consists of a set of process definitions, a method `main`, and possibly other, conventional program parts. Method `main` specifies the configurations and creates, sets up, and starts a set of processes. DistAlgo language constructs can be used in process definitions and method `main` and are implemented according to the semantics described; other, conventional program parts are implemented according to their conventional semantics.

Other language constructs. For other constructs, we use those in high-level object-oriented languages. We mostly use Python syntax (indentation for scoping, ':' for elaboration, '#' for comments, etc.) for succinctness, except with $v := exp$ for assignment and with a few conventions from Java (keyword `extends` for subclass, keyword `new` for object creation, and omission of `self`, the equivalent of `this` in Java, when there is no ambiguity) for ease of reading.

Example. Figure 2 shows Lamport's algorithm expressed in DistAlgo. The algorithm in Figure 1 corresponds to the body of `mutex` and the two `receive` definitions, 16 lines total; the rest of the program, 14 lines total, shows how the algorithm is used in an application. The execution of the application starts with method `main`, which configures the system to run (lines 25-30). Method `mutex` and the two `receive` definitions are executed when needed and follow the five rules in Figure 1 (lines 5-21). Recall that there is an implicit yield point before the `await` statement.

Note that Figure 2 is not meant to replace Figure 1, but to realize Figure 1 in a precisely executable manner. Figure 2 is meant to be high-level, compared with lower-level specifications and programs.

4 COMPILING TO EXECUTABLE PROGRAMS

Compilation generates code to create processes on the specified machine, take care of sending and receiving messages, and realize the specified configuration. In particular, it inserts appropriate message handlers at each yield point.

Processes and sending of messages. Process creation is compiled to creating a process on the specified or default machine and that has a private memory space for its fields. Each process is implemented using two threads: a main thread that executes the main flow of control of the process, and a helper thread that receives and enqueues messages sent to this process. Constructs involving a set of processes, such as `n new P`, can easily be compiled into loops.

Sending a message `m` to a process `p` is compiled into calls to a standard message passing API. If the sequence `sent` is used in the program, we also insert `sent.add(m to p)`. Calling a method on a remote process object is compiled into a remote method call.

Control flows and handling of received messages. Each yield point `l` is compiled into a call to a message handler method `l()` that updates the sequence `received`, if `received` is used in the program, and executes the bodies of the `receive` definitions whose `at` clause includes `l`. Precisely:

- Each `receive` definition is compiled into a method that takes a message `m` as argument, matches `m` against the message patterns in the `receive` clause, and if the matching succeeds, binds the variables in the matched pattern appropriately, and executes the statement in the body of this `receive` definition.
- Method `l()` compiled for yield point `l` does the following: for each un-handled message `m` from `p` to be handled, (1) execute `received.add(m from p)` if `received` is used in the program, (2) call the methods generated from the `receive` definitions whose `at` clause includes `l`, and (3) remove `m` from the message queue.

An `await` statement can be compiled into a synchronization using busy-waiting or blocking. We use blocking to wait until a new message arrives or the timeout specified in `await` is reached.

```
 1 class P extends process:
 2   def setup(s):
 3     self.s := s                # set of all other processes
 4     self.q := {}               # set of pending requests

 5   def mutex(task):             # run task with mutual exclusion
 6     -- request
 7     self.t := logical_time()                       # 1 in Fig 1
 8     send ('request', t, self) to s                 #
 9     q.add(('request', t, self))                    #
                                  # wait for own req < others in q
                                  #  and for acks from all in s
10     await each ('request', t2, p2) in q |          # 5 in Fig 1
                (t2,p2) != (t,self) implies (t,self) < (t2,p2)
11           and each p2 in s |                       #
                some received('ack', t2, =p2) | t2 > t
12     task()                     # critical section
13     -- release
14     q.del(('request', t, self))                    # 3 in Fig 1
15     send ('release', logical_time(), self) to s    #

16   receive ('request', t2, p2):                     # 2 in Fig 1
17     q.add(('request', t2, p2))                     #
18     send ('ack', logical_time(), self) to p2       #

19   receive ('release', _, p2):                      # 4 in Fig 1
20     for ('request', t2, =p2) in q:                 #
21       q.del(('request', t2, p2))                   #

22   def run():                 # main method for the process
       ...                      # do non-CS tasks of the process
23     def task(): ...          # define critical section task
24     mutex(task)              # run task with mutual exclusion
       ...                      # do non-CS tasks of the process

25 def main():                  # main method for the application
     ...                        # do other tasks of the application
26   configure channel = {reliable, fifo}
                                # use reliable and FIFO channel
27   configure clock = Lamport  # use Lamport clock
28   ps := 50 new P             # create 50 processes of P class
29   for p in ps: p.setup(ps-{p}) # pass to each process other processes
30   for p in ps: p.start()     # start the run method of each process
     ...                        # do other tasks of the application
```

Fig. 2. Original algorithm (lines 6-21) in a complete program in DistAlgo.

Configuration. Configuration options are taken into account during compilation in a straightforward way. Libraries and modules are used as much as possible. For example, when `fifo` or `reliable` channel is specified, the compiler can generate code that uses TCP sockets.

5 INCREMENTALIZING EXPENSIVE SYNCHRONIZATIONS

Incrementalization transforms expensive computations into efficient incremental computations with respect to updates to the values on which the computations depend. It (1) identifies all expensive queries, (2) determines all updates that may affect the query result, and (3) transforms the queries and updates into efficient incremental computations. Much of incrementalization has been studied previously, as discussed in Section 7.

The new method here is for (1) systematic handling of quantifications for synchronization as expensive queries, especially nested alternating universal and existential quantifications and quantifications containing complex order comparisons and (2) systematic handling of updates caused by all sending, receiving, and handling of messages in the same way as other updates in the program. The result is a drastic reduction of both time and space complexities.

Expensive computations using quantifications. Expensive computations in general involve repetition, including loops, recursive functions, comprehensions, aggregations, and quantifications over collections. Optimizations were studied most for loops, less for recursive functions, comprehensions, and aggregations, and least for quantifications, basically corresponding to how frequently these constructs have traditionally been used in programming. However, high-level queries are increasingly used in programming, and quantifications are dominantly used in writing synchronization conditions and assertions in specifications and very high-level programs. Unfortunately, if implemented straightforwardly, each quantification introduces a cost factor that is linear in the size of the collection quantified over.

Optimizing expensive quantifications in general is difficult, which is a main reason that they are not used in practical programs, not even logic programs, and programmers manually write more complex and error-prone code. The difficulty comes from expensive enumerations over collections and complex combinations of join conditions. We address this challenge by converting quantifications into aggregations that can be optimized systematically using previously studied methods. However, a quantification can be converted into multiple forms of aggregations. Which one to use depends on what kinds of updates must be handled, and on how the query can be incrementalized under those updates. Direct conversion of nested quantifications into nested aggregations can lead to much more complex incremental computation code and asymptotically worse time and space complexities for maintaining the intermediate query results.

Note that, for an existential quantification, we convert it to a more efficient aggregation if a witness is not needed; if a witness is needed, we incrementally compute the set of witnesses.

Converting quantifications to aggregations. We present all converted forms here and describe which forms to use after we discuss the updates that must be handled. The correctness of all rules presented have been proved, manually, using first-order logic and set theory. These rules ensure that the value of a resulting query expression equals the value of the original quantified expression.

Table 1 shows general rules for converting single quantifications into equivalent aggregations that use aggregation operator `count`. For converting universal quantifications, either rule 2 or 3 could be used. The choice does not affect the asymptotic cost, but only small constant factors: rule 2 requires maintaining `count s`, and rule 3 requires computing `not`; the latter is generally faster unless `count s` is already needed for other purposes, and is certainly faster when `not bexp` can be simplified, e.g., when `bexp` is a negation. The rules in Table 1 are general because `bexp` can be any Boolean expression, but they are for converting single quantifications. Nested quantifications can be converted one at a time from inside out, but the results may be much more complicated than necessary. For example,

```
each x in s | some x2 in s2 | bexp
```

would be converted using rule 1 to

```
each x in s | count {x2 in s2 | bexp} != 0
```

and then using rule 2 to

```
count {x in s | count {x2 in s2 | bexp} != 0} = count s
```

A simpler conversion is possible for this example, using a rule in Table 2, described next.

Table 1. Rules for converting single quantifications.

	Quantification	Aggregation
1	`some x in s | bexp`	`count {x in s | bexp} != 0`
2	`each x in s | bexp`	`count {x in s | bexp} = count s`
3		`count {x in s | not bexp} = 0`

Table 2 shows general rules for converting nested quantifications into equivalent, but non-nested, aggregations that use aggregation operator `count`. These rules yield much simpler results than repeated use of the rules in Table 1. For example, rule 2 in this table yields a much simpler result than using two rules in Table 1 in the previous example. More significantly, rules 1, 4, and 5 generalize to any number of the same quantifier, and rules 2 and 3 generalize to any number of quantifiers with one alternation. We have not encountered more complicated quantifications than these in the algorithms we found. It is well known that more than one alternation is rarely used, so commonly used quantifications can all be converted to non-nested aggregations. For example, in twelve different algorithms expressed in DistAlgo [56], there are a total of 50 quantifications but no occurrence of more than one alternation.

Table 2. Rules for converting nested quantifications.

	Nested Quantifications	Aggregation
1	`some x in s | some x2 in s2 | bexp`	`count {(x,x2): x in s, x2 in s2 | bexp} != 0`
2	`each x in s | some x2 in s2 | bexp`	`count {x: x in s, x2 in s2 | bexp} = count s`
3	`some x in s | each x2 in s2 | bexp`	`count {x: x in s, x2 in s2 | not bexp} != count s`
4	`each x in s | each x2 in s2 | bexp`	`count {(x,x2): x in s, x2 in s2 | bexp} = count {(x,x2): x in s, x2 in s2}`
5		`count {(x,x2): x in s, x2 in s2 | not bexp} = 0`

Table 3 shows general rules for converting single quantifications with a single order comparison, for any linear order, into equivalent queries that use aggregation operators `max` and `min`. These rules are useful because `max` and `min` can in general be maintained incrementally in $O(\log n)$ time with $O(n)$ space overhead. Additionally, when there are only element additions, `max` and `min` can be maintained most efficiently in $O(1)$ time and space.

Table 4 shows general rules for decomposing Boolean combinations of conditions in quantifications, to obtain quantifications with simpler conditions. In particular, Boolean combinations of order comparisons and other conditions can be transformed to extract quantifications each with a single order comparison, so the rules in Table 3 can be applied, and Boolean combinations of inner quantifications and other conditions can be transformed to extract directly nested quantifications, so the rules in Table 2 can be applied. For example,

```
each x in s | bexp implies y < x
```

can be converted using rule 8 in Table 4 to

Table 3. Rules for single quantified order comparison.

	Existential	Aggregation
1	`some x in s | y <= x`	`s != {} and y <= max s`
2	`some x in s | x >= y`	
3	`some x in s | y >= x`	`s != {} and y >= min s`
4	`some x in s | x <= y`	
5	`some x in s | y < x`	`s != {} and y < max s`
6	`some x in s | x > y`	
7	`some x in s | y > x`	`s != {} and y > min s`
8	`some x in s | x < y`	
	Universal	**Aggregation**
9	`each x in s | y <= x`	`s = {} or y <= min s`
10	`each x in s | x >= y`	
11	`each x in s | y >= x`	`s = {} or y >= max s`
12	`each x in s | x <= y`	
13	`each x in s | y < x`	`s = {} or y < min s`
14	`each x in s | x > y`	
15	`each x in s | y > x`	`s = {} or y > max s`
16	`each x in s | x < y`	

`each x in {x in s | bexp} | y < x`

which can then be converted using rule 13 of Table 3 to

`{x in s | bexp} = {} or y < min {x in s | bexp}`

Table 4. Rules for decomposing conditions to extract quantified comparisons.

	Quantification	Decomposed Quantifications
1	`some x in s | not e`	`not each x in s | e`
2	`some x in s | e1 and e2`	`some x in {x in s | e1} | e2`
3	`some x in s | e1 or e2`	`(some x in s | e1) or (some x in s | e2)`
4	`some x in s | e1 implies e2`	`(some x in s | not e1) or (some x in s | e2)`
5	`each x in s | not e`	`not some x in s | e`
6	`each x in s | e1 and e2`	`(each x in s | e1) and (each x in s | e2)`
7	`each x in s | e1 or e2`	`each x in {x in s | not e1} | e2`
8	`each x in s | e1 implies e2`	`each x in {x in s | e1} | e2`

Updates caused by message passing. Recall that the parameters of a query are variables in the query whose values are bound before the query. Updates that may affect the query result include not only updates to the query parameters but also updates to the objects and collections reachable from the parameter values. The most basic updates are assignments to query parameters, `v := exp`, where `v` is a query parameter. Other updates are to objects and collections used in the query. For objects, all updates can be expressed as field assignments, `o.f := exp`. For collections, all updates can be expressed as initialization to empty and element additions and removals, `s.add(x)` and `s.del(x)`.

For distributed algorithms, a distinct class of important updates are caused by message passing. Updates are caused in two ways:

(1) Sending and receiving messages updates the sequences `sent` and `received`, respectively. Before incrementalization, code is generated, as described in Section 4, to explicitly perform these updates.
(2) Handling of messages by code in `receive` definitions updates variables that are parameters of the queries for computing synchronization conditions, or that are used to compute the values of these parameters.

Once these are established, updates can be determined using previously studied analysis methods, e.g., [33, 53].

Incremental computation. Given expensive queries and updates to the query parameters, efficient incremental computations can be derived for large classes of queries and updates based on the language constructs used in them or by using a library of rules built on existing data structures [50, 53, 57, 65].

For aggregations converted from quantifications, algebraic properties of the aggregation operators are exploited to efficiently handle possible updates. In particular, each resulting aggregate query result can be obtained in $O(1)$ time and incrementally maintained in $O(1)$ time per update to the sets maintained and affected plus the time for evaluating the conditions in the aggregation once per update. The total maintenance time at each element addition or deletion to a query parameter is at least a linear factor smaller than computing the query result from scratch. Additionally, if aggregation operators `max` and `min` are used and there are only element additions, the space overhead is $O(1)$. Note that if `max` and `min` are used naively when there are element deletions, there may be an unnecessary overhead of $O(n)$ space and $O(\log n)$ maintenance time per update from using more sophisticated data structures to maintain the `max` or `min` under element deletion [21, 84, 85].

Incremental computation improves time complexity only if the total time of repeated expensive queries is larger than that of repeated incremental maintenance. This is generally true for incrementalizing expensive synchronization conditions because (1) expensive queries in the synchronization conditions need to be evaluated repeatedly at each relevant update to the message history, until the condition becomes true, and (2) incremental maintenance at each such update is at least a linear factor faster for single message updates and no slower generally than computing from scratch.

To allow the most efficient incremental computation under all given updates, our method transforms each top-level quantification as follows:

- For non-nested quantifications, if the conditions contain no order comparisons or there are deletions from the sets or sequences whose elements are compared, the rules in Table 1 are used. The space overhead is linear in the sizes of the sets maintained and being aggregated over.
- For non-nested quantifications, if the conditions contain order comparisons and there are only additions to the sets or sequences whose elements are compared, the rules in Table 4 are used to extract single quantified order comparisons, and then the rules in Table 3 are used to convert the extracted quantifications. In this case, the space overhead is reduced to constant.
- For nested quantifications with one level of nesting, the rules in Table 4 are used to extract directly nested quantifications, and then the rules in Table 2 are used. If the resulting incremental maintenance has constant-time overhead maintaining a linear-space structure, we are done. If it is linear-time overhead maintaining a quadratic-space structure, and if the conditions contain order comparisons, then the rules in Table 4 are used to extract single

quantified order comparisons, and then the rules in Table 3 are used. This can reduce the overhead to logarithmic time and linear space.

- In general, multiple ways of conversion may be possible, besides small constant-factor differences between rules 2 and 3 in Table 1 and rules 4 and 5 in Table 2. In particular, for nested quantifications with two or more alternations, one must choose which two alternating quantifiers to transform first, using rule 2 or 3 in Table 2. We have not encountered such queries and have not studied this aspect further. Our general method is to transform in all ways possible, obtain the time and space complexities for each result, and choose one with the best time and then space. Complexities are calculated using the cost model of the set operations given in Section 3. The number of possible ways is exponential in the worst case in the size of the query, but the query size is usually a small constant.

Table 5 summarizes well-known incremental computation methods for these aggregate queries. The methods are expressed as incrementalization rules: if a query in the program matches the query form in the table, and each update to a parameter of the query in the program matches an update form in the table, then transform the query into the corresponding replacement and insert at each update the corresponding maintenance; fresh variables are introduced for each different query to hold the query results or auxiliary data structures. In the third rule, data structure `ds` stores the argument set `s` of `max` and supports priority queue operations.

Table 5. Incrementalization rules for `count` and for `max`.

Query	Replacement	Cost
`count s`	`number`	$O(1)$
Updates	**Inserted Maintenance**	**Cost**
`s := {}`	`number := 0`	$O(1)$
`s.add(x)`	`if x not in s: number +:= 1`	$O(1)$
`s.del(x)`	`if x in s: number -:= 1`	$O(1)$

Query	Replacement	Cost
`max s`	`maximum`	$O(1)$
Updates	**Inserted Maintenance**	**Cost**
`s := {x}`	`maximum := x`	$O(1)$
`s.add(x)`	`if x > maximum: maximum := x`	$O(1)$

Query	Replacement	Cost
`max s`	`ds.max()`	$O(1)$
Updates	**Inserted Maintenance**	**Cost**
`s := {}`	`ds := new DS()`	$O(1)$
`s := {x}`	`ds := new DS(); ds.add(x)`	$O(1)$
`s.add(x)`	`if x not in s: ds.add(x)`	$O(\log \lvert s \rvert)$
`s.del(x)`	`if x in s: ds.del(x)`	$O(\log \lvert s \rvert)$

The overall incrementalization algorithm [53, 57, 65] introduces new variables to store the results of expensive queries and subqueries, as well as appropriate additional values, forming a set of invariants, transforms the queries and subqueries to use the stored query results and additional values, and transforms updates to query parameters to also do incremental maintenance of the stored query results and additional values.

In particular, if queries are nested, inner queries are transformed before outer queries. Note that a comprehension such as {x in s | bexp} is incrementalized with respect to changes to parameters of Boolean expression bexp as well as addition and removal of elements of s; if bexp contains nested subqueries, then after the subqueries are transformed, incremental maintenance of their query results become additional updates to the enclosing query.

At the end, variables and computations that are dead in the transformed program are eliminated. In particular, sequences received and sent will be eliminated when appropriate, because queries using them have been compiled into message handlers that only store and maintain values needed for incremental evaluation of the synchronization conditions.

Example. In the program in Figure 2, three quantifications are used in the synchronization condition in the await statement, and two of them are nested. The condition is copied below, except that ('ack',t2,=p2) in received is used in place of received('ack',t2,=p2).

```
each ('request', t2, p2) in q |
  (t2,p2) != (t,self) implies (t,self) < (t2,p2)
and each p2 in s |
  some ('ack', t2, =p2) in received | t2 > t
```

Converting quantifications into aggregations as described using Tables 1 through 4 proceeds as follows. In the first conjunct, the universal quantification is converted using rule 2 or 3 in Table 1, because it contains an order comparison with elements of q and there are element deletions from q; rule 3 is used here because it is slightly simpler after the negated condition is simplified. In the second conjunct, the nested quantification is converted using rule 2 in Table 2. The resulting expression is:

```
count {('request', t2, p2) in q |
        (t,self) > (t2,p2)}  =  0
and
count {p2: p2 in s, ('ack', t2, p2) in received |
        t2 > t}  =  count s
```

Updates to parameters of the first conjunct are additions and removals of requests to and from q, and also assignment to t. Updates to parameters of the second conjunct are additions of ack messages to received, and assignment to t, after the initial assignment to s.

Incremental computation [50, 53, 57, 65] introduces variables to store the values of all three aggregations in the converted query, transforms the aggregations to use the introduced variables, and incrementally maintains the stored values at each of the updates, as follows, yielding Figure 3.

- For the first conjunct, store the set value and the count value in two variables, say earlier and number1, respectively, so first conjunct becomes number1 = 0; when t is assigned a new value, let earlier be q and let number1 be its size, taking $O(|\texttt{earlier}|)$ time, amortized to $O(1)$ time when each request in earlier is served; when a request is added to q, if t is defined and (t,self) > (t2,p2) holds, add the request to earlier and increment number1 by 1, taking $O(1)$ time; similarly for deletion from q. A test of definedness, here t != undefined, is inserted for any variable that might not be defined in the scope of the maintenance code.

  Note that when ('request',t,self) in particular is added to or removed from q, earlier and number1 are not updated, because (t,self) > (t,self) is trivially false.
- For the second conjunct, store the set value and the two count values in three variables, say responded, number2, and total, respectively, so the conjunct becomes number2 = total; when s is initialized in setup, assign total the size of s, taking $O(|\texttt{s}|)$ time, done only once

for each process; when t is assigned a new value, let responded be {}, and let number2 be 0, taking $O(1)$ time; when an ack message is added to received, if the associated conditions hold, increment number2 by 1, taking $O(1)$ time. A test of definedness of t is omitted in the maintenance for receiving ack messages, because t is always defined there; this small optimization is incorporated in an incrementalization rule, but it could be done with a data-flow analysis that covers distributed data flows.

Note that incrementalization uses basic properties about primitives and libraries. These properties are incorporated in incrementalization rules. For the running example, the property used is that a call to `logical_time()` returns a timestamp larger than all existing timestamp values, and thus at the assignment to t in method mutex, we have that earlier is q and responded is {}. So, an incrementalization rule for maintaining earlier specifies that at update `t := logical_time()`, the maintenance is `earlier := q`; similarly for maintaining responded. These simplifications could be facilitated with data-flow analyses that determine variables holding logical times and sets holding certain element types. Incrementalization rules can use any program analysis results as conditions [50].

Figure 3 shows the optimized program after incrementalization of the synchronization condition on lines 10-11 in Figure 2. All lines with comments are new except that the synchronization condition in the await statement is simplified. The synchronization condition now takes $O(1)$ time, compared with $O(|s|^2)$ if computed from scratch. The trade-off is the amortized $O(1)$ time overhead at updates to t and q and on receiving of ack messages. Using based representation for sets [17, 34, 64], maintaining earlier and responded can each be done using one bit for each process.

Note that the sequence received used in the synchronization condition in Figure 2 is no longer used after incrementalization. All values needed for evaluating the synchronization condition are stored in new variables introduced: earlier, number1, responded, number2, and total, a drastic space improvement from unbounded for received to linear in the number of processes.

Example with naive use of aggregation operator min. Note that the resulting program in Figure 3 does not need to use a queue at all, even though a queue is used in the original description in Figure 1; the variable q is simply a set, and thus element addition and removal takes $O(1)$ time.

We show that if min is used naively, a more sophisticated data structure [21, 84, 85] supporting priority queue is needed, incurring an $O(\log n)$ time update instead of the $O(1)$ time in Figure 3. Additionally, for a query using min to be correct, special care must be taken to deal with the case when the argument to min is empty, because then min is undefined.

Consider the first conjunct in the synchronization condition in the await statement in Figure 2, copied below:

```
each ('request', t2, p2) in q |
  (t2,p2) != (t,self) implies (t,self) < (t2,p2)
```

One might have written the following instead, because it seems natural, especially if universal quantification is not supported:

```
(t,self) < min {(t2,p2): ('request', t2, p2) in q
                       | (t2,p2) != (t,self)}
```

However, that is incorrect, because the argument of min may be empty, in which case min is undefined.

Instead of resorting to commonly used special values, such as maxint, which is ad hoc and error prone in general, the empty case can be added as the first disjunct of a disjunction:

```
class P extends process:
  def setup(s):
    self.s := s
    self.total := count s          # total num of other processes
    self.q := {}

  def mutex(task):
    -- request
    self.t := logical_time()
    self.earlier := q              # set of pending earlier requests
    self.number1 := count earlier  # num of pending earlier requests
    self.responded := {}           # set of responded processes
    self.number2 := 0              # num of responded processes
    send ('request', t, self) to s
    q.add(('request', t, self))
    await number1 = 0
          and number2 = total      # use maintained results
    task()
    -- release
    q.del(('request', t, self))
    send ('release', logical_time(), self) to s

  receive ('request', t2, p2):
    if t != undefined:             # if t is defined
      if (t,self) > (t2,p2):       # comparison in conjunct 1
        if ('request',t2,p2) not in earlier:   # if not in earlier
          earlier.add(('request', t2, p2))     # add to earlier
          number1 +:= 1                        # increment number1
    q.add(('request', t2, p2))
    send ('ack', logical_time(), self) to p2

  receive ('ack', t2, p2):         # new message handler
    if t2 > t:                     # comparison in conjunct 2
      if p2 in s:                  # membership in conjunct 2
        if p2 not in responded:    # if not responded already
          responded.add(p2)        # add to responded
          number2 +:= 1            # increment number2

  receive ('release', _, p2):
    for ('request', t2, =p2) in q:
      if t != undefined:           # if t is defined
        if (t,self) > (t2,p2):     # comparison in conjunct 1
          if ('request',t2,p2) in earlier:     # if in earlier
            earlier.del(('request', t2, p2))   # delete from earlier
            number1 -:= 1                      # decrement number1
      q.del(('request', t2, p2))
```

Fig. 3. Optimized program after incrementalization. Definitions of run and main are as in Figure 2.

```
{(t2,p2): ('request', t2, p2) in q
        | (t2,p2) != (t,self)} = {}
or
(t,self) < min {(t2,p2): ('request', t2, p2) in q
                       | (t2,p2) != (t,self)}
```

In fact, the original universal quantification in the first conjunct in the `await` statement can be converted exactly to this disjunction by using rule 8 in Table 4 and then rule 13 in Table 3. Our method does not consider this conversion because it leads to a worse resulting program.

Figure 4 shows the resulting program after incrementalization of the synchronization condition that uses the disjunction above, where `ds` stores the argument set of `min` and supports priority queue operations. All commented lines are new compared to Figure 2 except that the synchronization condition in the `await` statement is simplified. The program appears shorter than Figure 3 because the long complex code for maintaining the data structure `ds` is not included; it is in fact similar to Figure 3 except that `ds` is used and maintained instead of `earlier` and `number1`.

The program in Figure 4 is still a drastic improvement over the original program in Figure 2, with the synchronization condition reduced to $O(1)$ time and with `received` removed, just as in Figure 3. The difference is that maintaining `ds` for incrementalizing `min` under element addition to and deletion from `q` takes $O(\log |s|)$ time, as opposed to $O(1)$ time for maintaining `earlier` and `number1` in Figure 3.

Simplifications to the original algorithm. Consider the original algorithm in Figure 2. Note that incrementalization determined that there is no need for a process to update auxiliary values for its own request, in both Figures 3 and 4. Based on this, we discovered, manually, that updates to `q` for a process's own request do not affect the two uses of `q`, on lines 9 and 35, in Figure 3 and the only use of `q`, on line 30, in Figure 4. So we can remove them in Figures 3 and 4. In addition, we can remove them on lines 9 and 14 in Figure 2 and remove the test `(t2,p2) != (t,self)`, which becomes always true, in the synchronization condition, yielding a simplified original algorithm.

Furthermore, note that the remaining updates to `q` in Figure 2 merely maintain pending requests by others, so we can remove lines 4, 17, 20, 21, and the entire `receive` definition for `release` messages, by using, for the first conjunct in the `await` statement,

```
each received('request', t2, p2) |
  not (some received('release', t3, =p2) | t3 > t2)
  implies (t,self) < (t2,p2)
```

Figure 5 shows the resulting simplified algorithm. Incrementalizing this program yields essentially the same programs as in Figures 3 and 4, except that it needs to use the property that when a message is added to `received`, all messages from the same process in `received` have a smaller timestamp. This property follows from the use of logical clock and FIFO channels. The incrementalization rules for maintaining the result of the new condition incorporate this property in a similar way as described for Figure 3, except it could be facilitated with also a data-flow analysis that determines the component of a received message holding the sender of the message.

6 IMPLEMENTATION AND EXPERIMENTS

We have developed a prototype implementation of the compiler and optimizations for DistAlgo and evaluated it in implementing a set of well-known distributed algorithms, as described previously [56]. We have also used DistAlgo in teaching distributed algorithms and distributed systems, and students used the language and system in programming assignments and course projects. We summarize

```
1 class P extends process:
2   def setup(s):
3     self.s := s
4     self.total := count s          # total num of other processes
5     self.q := {}
6     self.ds := new DS()            # data structure for maintaining
                                     #  requests by other processes
7   def mutex(task):
8     -- request
9     self.t := logical_time()
10    self.responded := {}           # set of responded processes
11    self.number := 0               # num of responded processes
12    send ('request', t, self) to s
13    q.add(('request', t, self))
14    await (ds.is_empty() or (t,self) < ds.min())
            and number = total       # use maintained results
15    task()
16    -- release
17    q.del(('request', t, self))
18    send ('release', logical_time(), self) to s

19  receive ('request', t2, p2):
20    ds.add((t2,p2))                # add to data structure
21    q.add(('request', t2, p2))
22    send ('ack', logical_time(), self) to p2

23  receive ('ack', t2, p2):         # new message handler
24    if t2 > t:                     # comparison in conjunct 2
25      if p2 in s:                  # membership in conjunct 2
26        if p2 not in responded:    # if not responded already
27          responded.add(p2)        # add to responded
28          number +:= 1             # increment number

29  receive ('release', _, p2):
30    for ('request', t2, =p2) in q:
31      ds.del((t2,p2))              # delete from data structure
32      q.del(('request', t2, p2))
```

Fig. 4. Optimized program with use of `min` after incrementalization. Definitions of `run` and `main` are as in Figure 2.

results from the former and describe experience with the latter, after an overview and update about the implementation.

Our DistAlgo implementation takes DistAlgo programs written in extended Python, applies analyses and optimizations, especially to the high-level queries, and generates executable Python code. It optionally interfaces with an incrementalizer to apply incrementalization before generating code. Applying incrementalization uses the methods and implementation from previous work: a library of incrementalization rules was developed, manually but mostly following a systematic method [53, 57], and applied automatically using InvTS [33, 50]. A set of heuristics are currently used to select the best program generated from incrementalizing differently converted aggregations.

```
1 class P extends process:
2   def setup(s):
3     self.s := s

4   def mutex(task):
5     -- request
6     self.t := logical_time()
7     send ('request', t, self) to s
8     await each received('request', t2, p2) |
              not (some received('release', t3, =p2) | t3 > t2)
              implies (t,self) < (t2,p2)
9           and each p2 in s |
              some received('ack', t2, =p2) | t2 > t
10    task()
11    -- release
12    send ('release', logical_time(), self) to s

13  receive ('request', _, p2):
14    send ('ack', logical_time(), self) to p2
```

Fig. 5. Simplified algorithm. Definitions of `run` and `main` are as in Figure 2.

A more extensive implementation of DistAlgo than the first prototype [56] has been released and is being gradually improved [25]. Improved methods and implementation for incrementalization are also being developed [49], to replace manually written incrementalization rules, and to better select the best transformed programs.

Evaluation in implementing distributed algorithms. We have used DistAlgo to implement a variety of well-known distributed algorithms, including twelve different algorithms for distributed mutual exclusion, leader election, and atomic commit, as well as Paxos, Byzantine Paxos, and multi-Paxos, as summarized previously [56]; results of evaluation using these programs are as follows:

- DistAlgo programs are consistently small, ranging from 22 to 160 lines, and are much smaller than specifications or programs written in other languages, mostly 1/2 to 1/5 of the size; also we were able to find only a few of these algorithms written in other languages. Our own best effort to write Lamport's distributed mutual exclusion in programming languages resulted in 272 lines in C, 216 lines in Java, 122 lines in Python, and 99 lines in Erlang, compared with 32 lines in DistAlgo.
- Compilation times without incrementalization are all under 0.05 seconds on an Intel Core-i7 2600K CPU with 16GB of memory; and incrementalization times are all under 30 seconds. Generated code size ranges from 1395 to 1606 lines of Python, including 1300 lines of fixed library code.
- Execution time and space confirm the analyzed asymptotic time and space complexities. For example, for Lamport's distributed mutual exclusion, total CPU time is linear in the number of processes for the incrementalized program, but superlinear for the original program; for a fixed number of processes, the memory usage is constant for the incremental program, but grows linearly with the number of requests for the original program.
- Compared with running times of our best, manually written programs in programming languages, all running on a single machine, our generated DistAlgo takes about twice as

long as our Python version, which takes about twice as long as our Java version, which takes about twice as long as our C version, which takes about four times as long as our Erlang version.

Python is well known to be slow compared Java and C, and we have not focused on optimizing constant factors. Erlang is significantly faster than C and the rest because of its use of light-weight threads to implement processes that is facilitated by its being a functional language. However, among all our programs for Lamport's distributed mutual exclusion, Erlang is the only one besides un-incrementalized DistAlgo whose memory usage for a fixed number of processes grows linearly with the number of requests.

Programming distributed algorithms at a high level has also allowed us to discover several improvements to correctness and efficiency aspects of some of the algorithms [55]. For example, in the pseudocode for multi-Paxos [82], in process `Commander`, waiting for `p2b` messages containing ballot `b` from a majority of `acceptors` is expressed by starting with a `waitfor` set initialized to `acceptors` and then, in a `for ever` loop, repeatedly updating `waitfor` and testing `|waitfor| < |acceptors|/2` as each `p2b` message containing ballot `b` arrives. The test is incorrect if implemented directly in commonly used languages such as Java, and even Python until Python 3, because `/` is integer division, which discards any fractional part; for example, test `1 < 3/2` becomes `false` but should be `true`. In DistAlgo, the entire code can simply be written as

```
await count {a: received ('p2b',=b) from a} > (count acceptors)/2
```

using the standard majority test, and it is correct whether `/` is for integer or float.

Experience in teaching distributed algorithms. DistAlgo has also helped us tremendously in teaching distributed algorithms, because it makes complex algorithms completely clear, precise, and directly executable. Students learn DistAlgo quickly through even a small programming assignment, despite that most did not know Python before, thanks to the power and clarity of Python.

In particular, students in distributed systems courses have used DistAlgo in dozens of course projects, implementing the core of network protocols and distributed graph algorithms [59]; distributed coordination services Chubby [16] and Zookeeper [38]; distributed hash tables Kademlia [61], Chord [79], Pastry [74], Tapestry [87], and Dynamo [24]; distributed file systems GFS [32] and HDFS [78]; distributed databases Bigtable [19], Cassandra [42], and Megastore [12]; distributed processing platform MapReduce [23]; and others.

All distributed programming features were used extensively in students' programs—easy process creation and setup and sending of messages, high-level control flows with `receive` definitions as well as `await` for synchronization, and declarative configurations—with the exception of queries over message histories, because students had been trained in many courses to handle events imperatively; we have not evaluated incrementalization on students' programs, because execution efficiency has not been a problem. Overall, students' experience helps confirm that DistAlgo allows complex distributed algorithms and services to be implemented much more easily than commonly used languages such as C++ and Java. We summarize two specific instances below.

In a graduate class in Fall 2012, most of the 28 students initially planned to use Java or C++ for their course projects, because they were familiar with those and wanted to strengthen their experience of using them instead of using DistAlgo in implementing distributed systems. However, after doing one programming assignment using DistAlgo, all those students switched to DistAlgo for their course projects, except for one student, who had extensive experience with C++, including several years of internship at Microsoft Research programming distributed systems.

- This student wrote about 3000 lines of C++, compared to about 300 lines of DistAlgo written by several other students who chose the same project of implementing multi-Paxos

and several optimizations. Furthermore, his C++ program was incomplete, lacking some optimizations that other students' DistAgo programs included.

- The student did a re-implementation in DistAlgo quickly after the course[1], confirming that it took about 300 lines. His biggest surprise was that his C++ program was an order of magnitude slower than his DistAlgo program. After several weeks of debugging, he found that it was due to an improper use of some C++ library function.

The main contrast that the student concluded was the huge advantage of DistAlgo over C++ in ease of programming and program understanding, not to mention the unexpected performance advantage.

In a graduate class in Fall 2014, each team of two students first implemented a fault-tolerant banking service in two languages: DistAlgo and another language of their choice other than Python. We excluded Python as the other language, because implementing the same service in such closely related languages would be less educational. The service uses chain replication [83] to tolerate crash failures. The service offers only a few simple banking operations (get balance, deposit, withdrawal, intra-bank transfer, inter-bank transfer), so most of the code is devoted to distributed systems aspects. The numbers of teams that chose various other languages are: Java 15, C++ 3, Go 3, Erlang 2, Node.js 2, Elixir (a variant of Erlang) 1, JavaScript 1.

- In the last assignment, teams implemented an extension to the banking service in one language of their choice. 59% of the teams chose DistAlgo for this, even though most students (about 80%) did not know Python, and none knew DistAlgo, at the beginning of the class. In other words, a majority of students decided that implementation of this type of system is better in DistAlgo, even compared to languages with which they had more experience and that are more widely used.
- We asked each team to compare their experiences with the two languages. Teams consistently reported that development in DistAlgo was faster and easier than development in the other language (even though most students did not know Python before the project), and that the DistAlgo code was significantly shorter. It is no surprise that Java and C++ require more code, even when students used existing networking libraries, which they were encouraged to do. Comparison with Erlang and Go is more interesting, because they are high-level languages designed to support distributed programming. For the teams that chose Erlang, the average DistAlgo and Erlang code sizes, measured as non-empty non-comment line of code, are 586 and 1303, respectively. For the teams that chose Go, the average DistAlgo and Go code sizes are 465 and 1695, respectively.

7 RELATED WORK AND CONCLUSION

A wide spectrum of languages and notations have been used to describe distributed algorithms, e.g., [7, 29, 41, 44, 45, 59, 70–72, 81]. At one end, pseudocode with English is used, e.g., [41], which gives a high-level flow of the algorithms, but lacks the details and precision needed for a complete understanding. At the other end, state machine based specification languages are used, e.g., I/O automata [39, 59], which is completely precise, but uses low-level control flows that make it harder to write and understand the algorithms. There are also many notations in between these extremes, some being much more precise or completely precise while also giving a high-level control flow, e.g., Raynal's pseudocode [70–72] and Lamport's PlusCal [45]. However, all of these languages and notations lack concepts and mechanisms for building real distributed applications, and most of the languages are not executable.

[1]The student wanted to do research on DistAlgo and so was asked to re-implement his project in DistAlgo.

Many programming languages support programming of distributed algorithms and applications. Most support distributed programming through messaging libraries, ranging from relatively simple socket libraries to complex libraries such as MPI [62]. Many support Remote Procedure Call (RPC) or Remote Method Invocation (RMI), which allows a process to call a subroutine in another process without the programmer coding the details for this. Many also support asynchronous method invocation (AMI), which allows the caller to not block and get the reply later. Some programming languages, such as Erlang [26, 46], which has an actor-like model [2], have support for message passing and process management built into the language. There are also other well-studied languages for distributed programming, e.g., Argus [47], Lynx [76], SR [5], Concert/C [8], and Emerald [15]. These languages all lack constructs for expressing control flows and complex synchronization conditions at a much higher level; such high-level constructs are extremely difficult to implement efficiently. DistAlgo's construct for declaratively and precisely specifying yield points for handling received messages is a new feature that we have not seen in other languages. So is DistAlgo's support of history variables in high-level synchronization conditions in non-deterministic `await` with timeout in a programming language. Our simple combination of synchronous `await` and asynchronous `receive` allows distributed algorithms to be expressed easily and clearly.

There has been much work on producing executable implementations from formal specifications, e.g., from process algebras [37], I/O automata [31], Unity [35], and Seuss [40], as well as from more recently proposed high-level languages for distributed algorithms, e.g., Datalog-based languages Meld [6], Overlog [4], and Bloom [13], a Prolog-based language DAHL [58], and a logic-based language EventML [14, 67]. An operational semantics was studied recently for a variant of Meld, called Linear Meld, that allows updates to be encoded more conveniently than Meld by using linear logic [22]. Compilation of DistAlgo to executable implementations is easy because it is designed to be so and DistAlgo is given an operational semantics. High-level queries and quantifications used for synchronization conditions can be compiled into loops straightforwardly, but they may be extremely inefficient. None of these prior works study powerful optimizations of quantifications. Efficiency concern is a main reason that similar high-level language constructs, whether for queries or assertions, are rarely used, if supported at all, in commonly used languages.

Incrementalization has been studied extensively, e.g., [48, 69], both for doing it systematically based on languages, and in applying it in an ad hoc fashion to specific problems. However, all systematic incrementalization methods based on languages have been for centralized sequential programs, e.g., for loops [3, 30, 54], set languages [36, 57, 65], recursive functions [1, 51, 68], logic rules [52, 75], and object-oriented languages [49, 53, 63, 73]. This work is the first to extend incrementalization to distributed programs to support high-level synchronization conditions. This allows the large body of previous work on incrementalization, especially on sets and sequences, to be used for optimizing distributed programs.

Quantifications are the centerpiece of first-order logic, and are dominantly used in writing synchronization conditions and assertions in specifications, but there are few results on generating efficient implementations of them. In the database area, despite extensive work on efficient implementation of high-level queries, efficient implementation of quantification has only been studied in limited scope or for extremely restricted query forms, e.g., [9–11, 20]. In logic programming, handling of universal quantification is based on variants of brute-force Lloyd-Topor transformations, e.g., [28, 66]; even state-of-the-art logic programming systems, e.g., [80], do not support universal quantification. Our method is the first general and systematic method for incrementalizing arbitrary quantifications. Although they are much more challenging to optimize than set queries, our method combines a set of general transformations to transform them into aggregations that can be most efficiently incrementalized using the best previous methods.

To conclude, this article presents a powerful language and method for programming and optimizing distributed algorithms. There are many directions for future work, from formal verification on the theoretical side, to generating code in lower-level languages on the practical side, with many additional analyses and optimizations in between. In particular, a language with a high level of abstraction also faciliates formal verification, of not only the high-level programs, but also the generated efficient implementations when they are generated through systematic optimizations. Besides developing systematic optimizations, we have started to study formal verification of distributed algorithms [18] and their implementations by starting with their high-level, concise descriptions in DistAlgo.

APPENDIX

A SEMANTICS OF DISTALGO

We give an abstract syntax and operational semantics for a core language for DistAlgo. The operational semantics is a reduction semantics with evaluation contexts [77, 86].

A.1 Abstract Syntax

The abstract syntax is defined in Figures 6 and 7. We use some syntactic sugar in sample code, e.g., we use infix notation for some binary operators, such as `and` and `is`.

Notation.

- A symbol in the grammar is a terminal symbol if it starts with a lower-case letter.
- A symbol in the grammar is a non-terminal symbol if it starts with an upper-case letter.
- In each production, alternatives are separated by a linebreak.
- $*$ after a non-terminal means "0 or more occurrences".
- + after a non-terminal means "1 or more occurrences".
- $t\theta$ denotes the result of applying substitution θ to t. We represent substitutions as functions from variables to expressions.

Well-formedness requirements on programs.

(1) The top-level method in a program must be named `main`. It gets executed in an instance of the pre-defined `process` class when the program starts.
(2) Each label used in a `receive` definition must be the label of some statement that appears in the same class as the `receive` definition.
(3) Invocations of methods defined using `def` appear only in method call statements. Invocations of methods defined using `defun` appear only in method call expressions.

Constructs whose semantics is given by translation.

(1) Constructors for all classes, and `setup()` methods for process classes, are eliminated by translation into ordinary methods that assign to the fields of the objects.
(2) A method call or field assignment that does not explicitly specify the target object is translated into a method call or field assignment, respectively, on `self`.
(3) An `await` statement without an explicitly specified label—in other words, the associated label is the empty string—is translated into an `await` statement with an explicitly specified label, by generating a fresh label name ℓ, replacing the empty label in that `await` statement with ℓ, and inserting ℓ in every `at` clause in the class containing the `await` statement.
(4) The Boolean operators `and` and `each` are eliminated as follows: e_1 `and` e_2 is replaced with `not(not(`e_1`) or not(`e_2`))`, and `each` *iter* `|` e is replaced with `not(some` *iter* `|` `not(`e`))`.

Program ::= *Configuration ProcessClass* Method*
ProcessClass ::= `class` *ClassName* `extends` *ClassName*: *Method* ReceiveDef**

ReceiveDef ::= `receive` *ReceivePattern+* `at` *Label+* : *Statement*
`receive` *ReceivePattern+* : *Statement*

ReceivePattern ::= *Pattern* `from` *InstanceVariable*

Method ::= `def` *MethodName*(*Parameter**) *Statement*
`defun` *MethodName*(*Parameter**) *Expression*

Statement ::=
InstanceVariable := *Expression*
InstanceVariable := `new` *ClassName*
InstanceVariable := { *Pattern* : *Iterator** | *Expression* }
Statement ; *Statement*
`if` *Expression*: *Statement* `else`: *Statement*
`for` *Iterator*: *Statement*
`while` *Expression*: *Statement*
Expression.*MethodName*(*Expression**)
`send` *Tuple* `to` *Expression*
Label `await` *Expression* : *Statement AnotherAwaitClause**
Label `await` *Expression* : *Statement AnotherAwaitClause** `timeout` *Expression*
`skip`

Expression ::= *Literal*
Parameter
InstanceVariable
Tuple
Expression.*MethodName*(*Expression**)
UnaryOp(*Expression*)
BinaryOp(*Expression*,*Expression*)
`isinstance`(*Expression*,*ClassName*)
`and`(*Expression*,*Expression*) // conjunction (short-circuiting)
`or`(*Expression*,*Expression*) // disjunction (short-circuiting)
`each` *Iterator* | *Expression*
`some` *Iterator* | *Expression*

Tuple ::= (*Expression**)

Fig. 6. Abstract syntax, Part 1.

(5) An aggregation is eliminated by translation into a comprehension followed by a `for` loop that iterates over the set returned by the comprehension. The `for` loop updates an accumulator variable using the aggregation operator.
(6) Iterators containing tuple patterns are rewritten as iterators without tuple patterns, as follows.
   - Consider the existential quantification `some` $(e_1, \ldots, e_n)$ `in` s | b. Let x be a fresh variable. Let θ be the substitution that replaces e_i with `select`(x, i) for each i such that e_i is a variable not prefixed with "=". Let $\{j_1, \ldots, j_m\}$ contain the indices of the

```
UnaryOp ::= not          // Boolean negation
            isTuple      // test whether a value is a tuple
            len          // length of a tuple
BinaryOp ::= is          // identity-based equality
             plus        // sum
             select      // select(t,i) returns the i'th component of tuple t

Pattern ::= InstanceVariable
            TuplePattern

TuplePattern ::= (PatternElement*)

PatternElement ::= Literal
                   InstanceVariable
                   =InstanceVariable

Iterator ::= Pattern in Expression

AnotherAwaitClause ::= or Expression : Statement

Configuration ::= configuration ChannelOrder ChannelReliability ...
ChannelOrder ::= fifo
                 unordered
ChannelReliability ::= reliable
                       unreliable

ClassName ::= ...
MethodName ::= ...
Parameter ::= ...
InstanceVariable ::= Expression.Field
Field ::= ...
Label ::= ...
Literal ::= BooleanLiteral
            IntegerLiteral
            ...
BooleanLiteral ::= true
                   false
IntegerLiteral ::= ...
```

Fig. 7. Abstract syntax, Part 2. Ellipses ("...") are for common syntactic categories whose details are unimportant.

constants and the variables prefixed with "=" in $(e_1, \ldots, e_n)$. Let $\bar{e}_j$ denote e_j after removing the "=" prefix, if any. The quantification is rewritten as `some` x `in` s `|` `isTuple(`x`)` `and` `len(`x`)` `is` n `and` `(select(`x`,`j_1`), ..., select(`x`,`j_m`))` `is` $(\bar{e}_{j_1}, \ldots, \bar{e}_{j_m})$ `and` $b\theta$.

- Consider the loop `for` $(e_1, \ldots, e_n)$ `in` e `:` s. Let x and S be fresh variables. Let $\{i_1, \ldots, i_k\}$ contain the indices in $(e_1, \ldots, e_n)$ of variables not prefixed with "=". Let θ be the substitution that replaces e_i with `select(`x`,`i`)` for each i in $\{i_1, \ldots, i_k\}$. Let $\{j_1, \ldots, j_m\}$ contain the indices in $(e_1, \ldots, e_n)$ of the constants and the variables prefixed with "=". Let $\bar{e}_j$ denote e_j after removing the "=" prefix, if any. Note that e may

denote a set or sequence, and duplicate bindings for the tuple of variables $(e_{i_1}, \ldots, e_{i_k})$ are filtered out if e is a set but not if e is a sequence. The loop is rewritten as the code in Figure 8.

```
S := e
if isinstance(S,set):
  S := { x : x in S | isTuple(x) and len(x) is n
       and (select(x,j1), ..., select(x,jm)) is (ē_j1, ..., ē_jm) }
  for x in S:
    sθ
else:  // S is a sequence
  for x in S:
    if (isTuple(x) and len(x) is n
        and (select(x,j1), ..., select(x,jm)) is (ē_j1, ..., ē_jm):
      sθ
    else:
      skip
```

Fig. 8. Translation of for loop to eliminate tuple pattern.

(7) Comprehensions in which some variables are prefixed with = are translated into comprehensions without such prefixing. Specifically, for a variable x prefixed with = in a comprehension, replace occurrences of =x in the comprehension with occurrences of a fresh variable y, and add the conjunct `y is x` to the Boolean condition.

(8) Comprehensions are statically eliminated as follows. The comprehension x `:= { e | ` x_1 `in` e_1`, ...,` x_n `in` e_n `| b }`, where each x_i is a pattern, is replaced with

```
x := new set
for x1 in e1:
  ...
    for xn in en:
      if b:
        x.add(e)
```

(9) Wildcards are eliminated from tuple patterns by replacing each occurrence of wildcard with a fresh variable.

(10) Remote method invocation, i.e., invocation of a method on another process after that process has been started, is translated into message communication.

Notes.

(1) *ClassName* must include process. process is a pre-defined class; it should not be defined explicitly. process has fields sent and received, and it has a method start.

(2) The grammar allows receive definitions to appear in classes that do not extend process, but such receive definitions are useless, so it would be reasonable to make them illegal.

(3) The grammar does not allow labels on statements other than await. A label ℓ on a statement s other than await is treated as syntactic sugar for label ℓ on `await true : skip` followed by statement s.

(4) *ClassName* must include set and sequence. Sets and sequences are treated as objects, because they are mutable. These are predefined classes that should not be defined explicitly. Methods of set include add, del, contains, min, max, and size. Methods of sequence include add (which adds an element at the end of the sequence), contains, and length. We give the semantics explicitly for a few of these methods; the others are handled similarly.

(5) Tuples are treated as immutable values, not as mutable objects.
(6) All expressions are side-effect free. For simplicity, we treat quantifications as expressions, so existential quantifications do not have the side-effect of binding variables to a witness. Such existential quantifications could be added as a new form of statement.
(7) Object creation and comprehension are statements, not expressions, because they have side-effects. Comprehension has the side-effect of creating a new `set`.
(8) *Parameter* must include `self`. The values of method parameters cannot be updated (e.g., using assignment statements). For brevity, local variables of methods are omitted from the core language. Consequently, assignment is allowed only for instance variables.
(9) Semantically, the `for` loop copies the contents of a (mutable) sequence or set into an (immutable) tuple before iterating over it, to ensure that changes to the sequence or set by the loop body do not affect the iteration. An implementation could use optimizations to achieve this semantics without copying when possible.
(10) For brevity, among the standard arithmetic operations (`+`, `-`, `*`, etc.), we include only one representative operation in the abstract syntax and semantics; others are handled similarly.
(11) The semantics below does not model real-time, so timeouts in `await` statements are simply allowed to occur non-deterministically.
(12) We omit the concept of node (process location) from the semantics, and we omit the node argument of the constructor when creating instances of process classes, because process location does not affect other aspects of the semantics.
(13) We omit `configure handling` statements from the syntax. The semantics is for `configure handling = all`. Semantics for other `configure handling` options can easily be added.
(14) To support initialization of a process by its parent, a process can access fields of another process and invoke methods on another process before the latter process is started.
(15) We require that all messages be tuples. This is an inessential restriction; it slightly simplifies the specification of pattern matching for matching messages against patterns.
(16) A process's `sent` sequence contains pairs of the form (m, d), where m is a message sent by the process to destination d. A process's `received` sequence contains pairs of the form (m, s), where m is a message received by the process from sender s.

A.2 Semantic Domains

The semantic domains are defined in Figure 9.

Notation.

- D^* contains finite sequences of values from domain D.
- Set(D) contains finite sets of values from domain D.
- $D1 \rightharpoonup D2$ contains partial functions from D_1 to D_2. $dom(f)$ is the domain of a partial function f.

Notes.

- We require that *ProcessAddress* and *NonProcessAddress* be disjoint.
- For $a \in ProcessAddress$ and $h \in Heap$, $h(a)$ is the local heap of process a. For $a \in Address$ and $ht \in HeapType$, $ht(a)$ is the type of the object with address a. For convenience, we use a single (global) function for *HeapType* in the semantics, even though the information in that function is distributed in the same way as the heap itself in an implementation.
- The *MsgQueue* associated with a process by the last component of a state contains messages, paired with the sender, that have arrived at the process but have not yet been handled by matching `receive` definitions.

$$
\begin{array}{rcl}
\mathit{Bool} & = & \{\texttt{true}, \texttt{false}\} \\
\mathit{Int} & = & \ldots \\
\mathit{ProcessAddress} & = & \ldots \\
\mathit{NonProcessAddress} & = & \ldots \\
\mathit{Address} & = & \mathit{ProcessAddress} \cup \mathit{NonProcessAddress} \\
\mathit{Tuple} & = & \mathit{Val}^* \\
\mathit{Val} & = & \mathit{Bool} \cup \mathit{Int} \cup \mathit{Address} \cup \mathit{Tuple} \\
\mathit{SetOfVal} & = & \mathrm{Set}(\mathit{Val}) \\
\mathit{SeqOfVal} & = & \mathit{Val}^* \\
\mathit{Object} & = & (\mathit{Field} \rightharpoonup \mathit{Val}) \cup \mathit{SetOfVal} \cup \mathit{SeqOfVal} \\
\mathit{HeapType} & = & \mathit{Address} \rightharpoonup \mathit{ClassName} \\
\mathit{LocalHeap} & = & \mathit{Address} \rightharpoonup \mathit{Object} \\
\mathit{Heap} & = & \mathit{ProcessAddress} \rightharpoonup \mathit{LocalHeap} \\
\mathit{ChannelStates} & = & \mathit{ProcessAddress} \times \mathit{ProcessAddress} \rightharpoonup \mathit{Tuple}^* \\
\mathit{MsgQueue} & = & (\mathit{Tuple} \times \mathit{ProcessAddress})^* \\
\mathit{State} & = & (\mathit{ProcessAddress} \rightharpoonup \mathit{Statement}) \times \mathit{HeapType} \times \mathit{Heap} \times \mathit{ChannelStates} \\
 & & \times (\mathit{ProcessAddress} \rightharpoonup \mathit{MsgQueue})
\end{array}
$$

Fig. 9. Semantic domains. Ellipses are used for semantic domains of primitive values whose details are standard or unimportant.

A.3 Extended Abstract Syntax

Section A.1 defines the abstract syntax of programs that can be written by the user. Figure 10 extends the abstract syntax to include additional forms into which programs may evolve during evaluation. Only the new productions are shown here; all of the productions given above carry over unchanged.

$$
\begin{array}{rcl}
\mathit{Expression} & ::= & \mathit{Address} \\
 & & \mathit{Address.Field} \\
 & & \\
\mathit{Statement} & ::= & \texttt{for}\ \mathit{Variable}\ \texttt{intuple}\ \mathit{Tuple}\texttt{:}\ \mathit{Statement}
\end{array}
$$

Fig. 10. Extensions to the abstract syntax.

The statement `for` v `intuple` t`:` s iterates over the elements of tuple t, in the obvious way.

A.4 Evaluation Contexts

Evaluation contexts, also called reduction contexts, are used to identify the next part of an expression or statement to be evaluated. An evaluation context is an expression or statement with a hole, denoted `[]`, in place of the next sub-expression or sub-statement to be evaluated. Evaluation contexts are defined in Figure 11.

$$
\begin{array}{l}
\mathit{Val} ::= \mathit{Literal} \\
\quad\quad \mathit{Address} \\
\quad\quad (\mathit{Val}^*) \\
C ::= [\,] \\
\quad\quad (\mathit{Val}^*, C, \mathit{Expression}^*) \\
\quad\quad C.\mathit{MethodName}(\mathit{Expression}^*) \\
\quad\quad \mathit{Address}.\mathit{MethodName}(\mathit{Val}^*, C, \mathit{Expression}^*) \\
\quad\quad \mathit{UnaryOp}(C) \\
\quad\quad \mathit{BinaryOp}(C, \mathit{Expression}) \\
\quad\quad \mathit{BinaryOp}(\mathit{Val}, C) \\
\quad\quad \texttt{isinstance}(C, \mathit{ClassName}) \\
\quad\quad \texttt{or}(C, \mathit{Expression}) \\
\quad\quad \texttt{some}\ \mathit{Pattern}\ \texttt{in}\ C\ \texttt{|}\ \mathit{Expression} \\
\quad\quad C.\mathit{Field} := \mathit{Expression} \\
\quad\quad \mathit{Address}.\mathit{Field} := C \\
\quad\quad \mathit{InstanceVariable} := C \\
\quad\quad C\ ;\ \mathit{Statement} \\
\quad\quad \texttt{if}\ C\texttt{:}\ \mathit{Statement}\ \texttt{else:}\ \mathit{Statement} \\
\quad\quad \texttt{for}\ \mathit{InstanceVariable}\ \texttt{in}\ C\texttt{:}\ \mathit{Statement} \\
\quad\quad \texttt{for}\ \mathit{InstanceVariable}\ \texttt{intuple}\ \mathit{Tuple}\texttt{:}\ C \\
\quad\quad \texttt{send}\ C\ \texttt{to}\ \mathit{Expression} \\
\quad\quad \texttt{send}\ \mathit{Val}\ \texttt{to}\ C \\
\quad\quad \texttt{await}\ \mathit{Expression}\ :\ \mathit{Statement}\ \mathit{AnotherAwaitClause}^*\ \texttt{timeout}\ C
\end{array}
$$

Fig. 11. Evaluation contexts.

A.5 Transition Relations

The transition relation for expressions has the form $ht : h \vdash e \longrightarrow e'$, where e and e' are expressions, $ht \in \mathit{HeapType}$, and $h \in \mathit{LocalHeap}$. The transition relation for statements has the form $\sigma \longrightarrow \sigma'$ where $\sigma \in \mathit{State}$ and $\sigma' \in \mathit{State}$.

Both transition relations are implicitly parameterized by the program, which is needed to look up method definitions and configuration information. The transition relation for expressions is defined in Figure 12. The transition relation for statements is defined in Figures 13–14.

Notation and auxiliary functions.

- In the transition rules, a matches an address; v matches a value (i.e., an element of Val); and ℓ matches a label.
- For an expression or statement e, $e[x := y]$ denotes e with all occurrences of x replaced with y.
- A function matches the pattern $f[x \longrightarrow y]$ iff $f(x)$ equals y. For example, in transition rules for statements, a function P in $\mathit{ProcessAddress} \rightharpoonup \mathit{Statement}$ matches $P[a \longrightarrow s]$ if P maps process address a to statement s.
- For a function f, $f[x := y]$ denotes the function that is the same as f except that it maps x to y.
- f_0 denotes the empty partial function, i.e., the partial function whose domain is the empty set.
- For a (partial) function f, $f \ominus a$ denotes the function that is the same as f except that it has no mapping for a.
- Sequences are denoted with angle brackets, e.g., $\langle 0, 1, 2 \rangle \in \mathit{Int}^*$.

- $s@t$ is the concatenation of sequences s and t.
- $first(s)$ is the first element of sequence s.
- $rest(s)$ is the sequence obtained by removing the first element of s.
- $length(s)$ is the length of sequence s.
- $extends(c_1, c_2)$ holds iff class c_1 is a descendant of class c_2 in the inheritance hierarchy.
- For $c \in ClassName$, $new(c)$ returns a new instance of c.

$$new(c) = \begin{cases} \{\} & \text{if } c = \texttt{set} \\ \langle\rangle & \text{if } c = \texttt{sequence} \\ f_0 & \text{otherwise} \end{cases}$$

- For $m \in MethodName$ and $c \in ClassName$, the relation $methodDef(c, m, def)$ holds iff (1) class c defines method m, and def is the definition of m in c, or (2) c does not define m, and def is the definition of m in the nearest ancestor of c in the inheritance hierarchy that defines m.
- For $h, \bar{h}, \bar{h}' \in LocalHeap$ and $ht, ht' \in HeapType$ and $v, \bar{v} \in Val$, the relation $isCopy(v, h, \bar{h}, ht, \bar{v}, \bar{h}', ht')$ holds iff (1) v is a value in a process with local heap h, i.e., addresses in v are evaluated with respect to h, (2) $\bar{v}$ is a copy of v for a process whose local heap was $\bar{h}$ before v was copied into it and whose local heap is $\bar{h}'$ after v is copied into it, i.e., $\bar{v}$ is the same as v except that, instead of referencing objects in h, it references newly created copies of those objects in $\bar{h}'$, and (3) $\bar{h}'$ and ht' are versions of $\bar{h}$ and ht updated to reflect the creation of those objects. As an exception, because process addresses are used as global identifiers, process addresses in v are copied unchanged into $\bar{v}$, and new copies of process objects are not created. We give auxiliary definitions and then a formal definition of *isCopy*.

  For $v \in Val$, let $addrs(v, h)$ denote the set of addresses that appear in v or in any objects or values reachable from v with respect to local heap h; formally,

$$\begin{aligned} & a \in addrs(v, h) \Leftrightarrow \\ & \quad (v \in Address \wedge v = a) \\ & \quad \vee (v \in dom(h) \wedge h(v) \in Field \rightharpoonup Val \wedge (\exists f \in dom(h(v)).\ a \in addrs(h(v)(f), h))) \\ & \quad \vee (v \in dom(h) \wedge h(v) \in SetOfVal \cup SeqOfVal \wedge (\exists v' \in h(v).\ a \in addrs(v', h))) \\ & \quad \vee (\exists v_1, \ldots, v_n \in Val.\ v = (v_1, \ldots, v_n) \wedge \exists i \in [1..n].\ a \in addrs(v_i, h)) \end{aligned}$$

  For $v, \bar{v} \in Val$ and $f \in Address \rightharpoonup Address$, the relation $subst(v, \bar{v}, f)$ holds iff v is obtained from $\bar{v}$ by replacing each occurrence of an address a in $dom(f)$ with $f(a)$ (informally, f maps addresses of new objects in $\bar{v}$ to addresses of corresponding old objects in v); formally,

$$\begin{aligned} subst(v, \bar{v}, f) \Leftrightarrow\ & (v \in Bool \cup Int \cup (Address \setminus dom(f)) \wedge \bar{v} = v) \\ & \vee (v \in dom(f) \wedge f(\bar{v}) = v) \\ & \vee (\exists v_1, \ldots, v_n, \bar{v}_1, \ldots, \bar{v}_n.\ v = (v_1, \ldots, v_n) \wedge \bar{v} = (\bar{v}_1, \ldots, \bar{v}_n) \\ & \quad \wedge (\forall i \in [1..n].\ subst(v_i, \bar{v}_i, f))) \end{aligned}$$

  Similarly, for $o, \bar{o} \in Object$ and $f \in Address \rightharpoonup Address$, the relation $subst(o, \bar{o}, f)$ holds iff o is obtained from $\bar{o}$ by replacing each occurrence of an address a in $dom(f)$ with $f(a)$. For sets S and S', let $S \xrightarrow{1-1} S'$ be the set of bijections between S and S'.

Finally, *isCopy* is defined as follows (intuitively, A contains the addresses of the newly allocated objects):

$$\begin{aligned}
&\mathit{isCopy}(v, h, \bar{h}, ht, \bar{v}, \bar{h}', ht') \Leftrightarrow \\
&\quad \exists A \subset \mathit{NonProcessAddress}.\ \exists f \in A \xrightarrow{1-1} (\mathit{addrs}(v, h) \setminus \mathit{ProcessAddress}). \\
&\qquad A \cap \mathit{dom}(ht) = \emptyset \wedge \mathit{dom}(ht') = \mathit{dom}(ht) \cup A \wedge \mathit{dom}(\bar{h}') = \mathit{dom}(\bar{h}) \cup A \\
&\qquad \wedge (\forall a \in \mathit{dom}(ht).\ ht'(a) = ht(a)) \wedge (\forall a \in \mathit{dom}(\bar{h}).\ \bar{h}'(a) = \bar{h}(a)) \\
&\qquad \wedge (\forall a \in A.\ ht'(a) = ht(f(a)) \wedge \mathit{subst}(h(a), \bar{h}'(a), f))
\end{aligned}$$

- For $m \in \mathit{Val}$, $a \in \mathit{ProcessAddress}$, $\ell \in \mathit{Label}$, $h \in \mathit{LocalHeap}$, and a `receive` definition d, if message m can be received from a at label ℓ by a process with local heap h using `receive` definition d, then $\mathit{matchRcvDef}(m, a, \ell, h, d)$ returns the appropriately instantiated body of d.

  We first define some auxiliary relations and functions. The relation $\mathit{matchesDefLbl}(d, \ell)$ holds iff `receive` definition d either lacks an `at` clause or has an `at` clause that includes ℓ. $\mathit{bound}(P)$ returns the set of variables that appear in pattern P prefixed with "=". $\mathit{vars}(P)$ returns the set of variables that appear in P. $\mathit{findSubstPat}(m, a, h, P\ \texttt{from}\ x)$ returns the substitution θ with domain $\mathit{vars}(P) \cup \{x\}$ such that $m = P\theta \wedge \theta(x) = a \wedge (\forall y \in \mathit{bound}(P).\ \theta(y) = h(y))$, if any, otherwise it returns $\perp$. $\mathit{findSubst}(m, a, h, d)$ returns $\mathit{findSubstPat}(m, a, h, P\ \texttt{from}\ x)$ for the first `receive` pattern $P\ \texttt{from}\ x$ in d such that $\mathit{findSubstPat}(m, a, h, P\ \texttt{from}\ x) \neq \perp$, if any, otherwise it returns $\perp$.

  If $\mathit{matchesDefLbl}(d, \ell) \wedge \mathit{findSubst}(m, a, h, d) \neq \perp$, then $\mathit{matchRcvDef}(m, a, \ell, h, d)$ returns $s\theta$, where s is the body of d (i.e., the statement that appears in d) and $\theta = \mathit{findSubst}(m, a, h, d)$, otherwise it returns $\perp$.
- For $m \in \mathit{Val}$, $a \in \mathit{ProcessAddress}$, $\ell \in \mathit{Label}$, $c \in \mathit{ClassName}$, and $h \in \mathit{LocalHeap}$, if message m can be received from a at label ℓ in class c by a process with local heap h, then $\mathit{receiveAtLabel}((m, a), \ell, c, h)$ returns a set of statements that should be executed when receiving m in that context.

  Specifically, if class c contains a `receive` definition d such that $\mathit{matchRcvDef}(m, a, \ell, h, d)$ is not $\perp$, then, letting $d_1, \ldots, d_n$ be the `receive` definitions in c such that $\mathit{matchRcvDef}(m, a, \ell, h, d_i)$ is not $\perp$, and letting $s_i = \mathit{matchRcvDef}(m, a, \ell, h, d_i)$, $\mathit{receiveAtLabel}((m, a), \ell, c, h)$ returns $\{s_1, \ldots, s_n\}$. Otherwise, $\mathit{receiveAtLabel}((m, a), \ell, c, h)$ returns $\emptyset$.

A.6 Executions

An execution is a sequence of transitions $\sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \cdots$ such that σ_0 is an initial state. The set of initial states is defined in Figure 15. Intuitively, a_p is the address of the initial process, a_r is the address of the `received` sequence of the initial process, and a_s is the address of the `sent` sequence of the initial process.

Informally, execution of the statement initially associated with a process may eventually (1) terminate (i.e., the statement associated with the process becomes `skip`, indicating that there is nothing left for the process to do), (2) get stuck (i.e., the statement associated with the process is not `skip`, and the process has no enabled transitions) due to an unsatisfied `await` statement or an error (e.g., the statement contains an expression that tries to select a component from a value that is not a tuple, or the statement contains an expression that tries to read the value of a non-existent field), or (3) run forever due to an infinite loop or infinite recursion.

// field access
$ht : h \vdash a.f \longrightarrow h(a)(f)$ if $a \in dom(h) \wedge f \in dom(h(a))$

// invoke method in user-defined class
$ht : h \vdash a.m(v_1, \ldots, v_n) \longrightarrow e[\texttt{self} := a, x_1 := v_1, \ldots, x_n := v_n]$
if $a \in dom(h) \wedge methodDef(ht(a), m, \texttt{defun}\ m(x_1, \ldots, x_n)\ e)$

// invoke method in pre-defined class (representative examples)
$ht : h \vdash a.\texttt{contains}(v_1) \longrightarrow \texttt{true}$ if $a \in dom(h) \wedge ht(a) = \texttt{set} \wedge v_1 \in h(a)$
$ht : h \vdash a.\texttt{contains}(v_1) \longrightarrow \texttt{false}$ if $a \in dom(h) \wedge ht(a) = \texttt{set} \wedge v_1 \notin h(a)$

// unary operations
$ht : h \vdash \texttt{not}(\texttt{true}) \longrightarrow \texttt{false}$
$ht : h \vdash \texttt{not}(\texttt{false}) \longrightarrow \texttt{true}$
$ht : h \vdash \texttt{isTuple}(v) \longrightarrow \texttt{true}$ if v is a tuple
$ht : h \vdash \texttt{isTuple}(v) \longrightarrow \texttt{false}$ if v is not a tuple
$ht : h \vdash \texttt{len}(v) \longrightarrow n$ if v is a tuple with n components

// binary operations
$ht : h \vdash \texttt{is}(v_1, v_2) \longrightarrow \texttt{true}$ if v_1 and v_2 are the same (identical) value

$ht : h \vdash \texttt{plus}(v_1, v_2) \longrightarrow v_3$ if $v_1 \in Int \wedge v_2 \in Int \wedge v_3 = v_1 + v_2$

$ht : h \vdash \texttt{select}(v_1, v_2) \longrightarrow v_3$
if $v_2 \in Int \wedge v_2 > 0 \wedge$ (v_1 is a tuple with at least v_2 components)
$\wedge$ (v_3 is the v_2'th component of v_1)

// isinstance
$ht : h \vdash \texttt{isinstance}(a, c) \longrightarrow \texttt{true}$ if $ht(a) = c$
$ht : h \vdash \texttt{isinstance}(a, c) \longrightarrow \texttt{false}$ if $ht(a) \neq c$

// disjunction
$ht : h \vdash \texttt{or}(\texttt{true}, e) \longrightarrow \texttt{true}$
$ht : h \vdash \texttt{or}(\texttt{false}, e) \longrightarrow e$

// existential quantification
$ht : h \vdash \texttt{some}\ x\ \texttt{in}\ a \mid e \longrightarrow e[x := v_1]\ \texttt{or}\ \cdots\ \texttt{or}\ e[x := v_n]$
if ($ht(a) = \texttt{sequence} \wedge h(a) = \langle v_1, \ldots, v_n \rangle$)
$\vee$ ($ht(a) = \texttt{set} \wedge \langle v_1, \ldots, v_n \rangle$ is a linearization of $h(a)$)

Fig. 12. Transition relation for expressions.

ACKNOWLEDGMENTS

We thank Michael Gorbovitski for supporting the use of InvTS for automatic incrementalization of DistAlgo programs. We are grateful to the following people for their helpful comments and discussions: Ken Birman, Andrew Black, Jon Brandvein, Wei Chen, Ernie Cohen, Mike Ferdman, John Field, Georges Gonthier, Leslie Lamport, Nancy Lynch, Lambert Meertens, Stephan Merz, Don Porter, Michel Raynal, John Reppy, Emin Gün Sirer, Doug Smith, Gene Stark, and Robbert van Renesse. We thank the anonymous reviewers for their detailed and helpful comments.

// field assignment
$(P[a \to a'.f := v], ht, h[a \to ha[a' \to o]], ch, mq)$
$\to (P[a := \texttt{skip}], ht, h[a := ha[a' := o[f := v]]], ch, mq)$

// object creation
$(P[a \to a'.f := \texttt{new}\ c], ht, h[a \to ha[a' \to o]], ch, mq)$
$\to (P[a := \texttt{skip}], ht[a' := c], h[a := ha[a' := o[f := a_c]], a_c := new(c)]], ch, mq)$
if $a_c \notin dom(ht) \wedge a_c \in Address \wedge (a_c \in ProcessAddress \Leftrightarrow extends(c, \texttt{process}))$

// sequential composition
$(P[a \to \texttt{skip}; s], ht, h, ch, mq) \to (P[a := s], ht, h, ch, mq)$

// conditional statement
$(P[a \to \texttt{if true} : s_1\ \texttt{else} : s_2], ht, h, ch, mq) \to (P[a := s_1], ht, h, ch, mq)$

$(P[a \to \texttt{if false} : s_1\ \texttt{else} : s_2], ht, h, ch, mq) \to (P[a := s_2], ht, h, ch, mq)$

// for loop
$(P[a \to \texttt{for}\ x\ \texttt{in}\ a' : s], ht, h, ch, mq) \to (P[a := \texttt{for}\ x\ \texttt{intuple}\ (v_1, \ldots, v_n) : s], ht, h, ch, mq)$
if $((ht(a) = \texttt{sequence} \wedge h(a)(a') = \langle v_1, \ldots, v_n \rangle)$
$\vee\ (ht(a) = \texttt{set} \wedge \langle v_1, \ldots, v_n \rangle$ is a linearization of $h(a)(a')))$

$(P[a \to \texttt{for}\ x\ \texttt{intuple}\ (v_1, \ldots, v_n) : s], ht, h, ch, mq)$
$\to (P[a := s[x := v_1]; \texttt{for}\ x\ \texttt{intuple}\ (v_2, \ldots, v_n) : s], ht, h, ch, mq)$

$(P[a \to \texttt{for}\ x\ \texttt{intuple}\ () : s], ht, h, ch, mq) \to (P[a := \texttt{skip}], ht, h, ch, mq)$

// while loop
$(P[a \to \texttt{while}\ e : s], ht, h, ch, mq) \to (P[a := \texttt{if}\ e : (s; \texttt{while}\ e : s)\ \texttt{else} : \texttt{skip}], ht, h, ch, mq)$

// invoke method in user-defined class
$(P[a \to a'.m(v_1, \ldots, v_n)], ht, h, ch, mq)$
$\to (P[a := s[\texttt{self} := a, x_1 := v_1, \ldots, x_n := v_n]], ht, h, ch, mq)$
if $a' \in dom(h(a))$
$\wedge\ ht(a') \notin \{\texttt{process}, \texttt{set}, \texttt{sequence}\} \wedge methodDef(ht(a'), m, \texttt{def}\ m(x_1, \ldots, x_n)\ s)$

// invoke method in pre-defined class (representative examples)

// `process.start` allocates a local heap and `sent` and `received` sequences for the new process,
// and moves the started process to the new local heap.
$(P[a \to a'.\texttt{start}()], ht, h[a \to ha[a' \to o], ch, mq)$
$\to (P[a := \texttt{skip}, a' := a'.\texttt{run}()], ht[a_s := \texttt{sequence}, a_r := \texttt{sequence}],$
$h[a := ha \ominus a', a' := f_0[a' \to o[\texttt{sent} := a_s, \texttt{received} := a_r], a_r := \langle\rangle, a_s := \langle\rangle]], ch, mq)$
if $extends(ht(a'), \texttt{process}) \wedge (ht(a')$ inherits `start` from `process`$) \wedge a_r \notin dom(ht) \wedge a_s \notin dom(ht)$
$\wedge\ a_r \in NonProcessAddress \wedge a_s \in NonProcessAddress$

$(P[a \to a'.\texttt{add}(v_1)], ht, h[a \to ha], ch, mq)$
$\to (P[a := \texttt{skip}], ht, h[a := ha[a' := ha(a') \cup \{v_1\}]], ch, mq)$
if $a' \in dom(ha) \wedge ht(a') = \texttt{set}$

$(P[a \to a'.\texttt{add}(v_1)], ht, h[a \to ha], ch, mq)$
$\to (P[a := \texttt{skip}], ht, h[a := ha[a' := ha(a') @ \langle v_1 \rangle]], ch, mq)$
if $a' \in dom(ha) \wedge ht(a') = \texttt{sequence}$

Fig. 13. Transition relation for statements, Part 1.

// send a message to one process. create copies of the message for the sender's sent sequence
// and the receiver.
$(P[a \to \texttt{send}\ v\ \texttt{to}\ a_2], ht, h[a \to ha, a_2 \to ha_2], ch, mq)$
$\to (P[a := \texttt{skip}], ht'', h[a := ha'[a_s := ha(a_s)@\langle(v_1, a_2)\rangle], a_2 := ha'_2], ch[(a, a_2) := ch((a, a_2))@\langle v_2\rangle], mq)$
if $a_2 \in ProcessAddress \wedge a_s = ha(a)(\texttt{sent}) \wedge isCopy(v, ha, ha, ht, v_1, ha', ht') \wedge isCopy(v, ha', ha_2, ht', v_2, ha'_2, ht'')$

// send to a set of processes
$(P[a \to \texttt{send}\ v\ \texttt{to}\ a'], ht, h[a \to ha], ch, mq)$
$\to (P[a := \texttt{for}\ x\ \texttt{in}\ a' \texttt{: send}\ v\ \texttt{to}\ x], ht, h[a := ha[a_s := ha(a_s)@\langle(v, a')\rangle]], ch, mq)$
if $ht(a') = \texttt{set} \wedge a_s = ha(a)(\texttt{sent}) \wedge (x$ is a fresh variable$)$

// message reordering
$(P, ht, h, ch[(a, a') \to q], mq) \to (P, ht, h, ch[(a, a') := q'], mq)$
if (channel order is unordered in the program configuration) $\wedge$ (q' is a permutation of q)

// message loss
$(P, ht, h, ch[(a, a') \to q], mq) \to (P, ht, h, ch[(a, a') := q'], mq)$
if (channel reliability is unreliable in the program configuration) $\wedge$ (q' is a subsequence of q)

// arrival of a message from process a at process a'. remove message from channel, and append
// (message, sender) pair to message queue.
$(P, ht, h, ch[(a, a') \to q], mq)$
$\to (P, ht, h, ch[(a, a') := rest(q)], mq[a' := mq(a')@\langle(first(q), a)\rangle])$
if $length(q) > 0$

// handle a message at a yield point. remove the (message, sender) pair from the message
// queue, append a copy to the received sequence, and prepare to run matching receive
// handlers associated with ℓ, if any. s has a label hence must be await.
$(P[a \to \ell\ s], ht, h[a \to ha], ch, mq[a \to q])$
$\to (P[a := s'[\texttt{self} := a]; \ell\ s], ht', h[a \to ha'[a_r \to ha(a_r)@\langle copy\rangle]], ch, mq[a := rest(q)])$
if $length(q) > 0 \wedge a_r = ha(a)(\texttt{received}) \wedge isCopy(first(q), ha, ha, ht, copy, ha', ht') \wedge receiveAtLabel(first(q), \ell, ht(a), ha') = S \wedge s'$ is a linearization of S

// await without timeout clause
$(P[a \to \ell\ \texttt{await}\ e_1{:}s_1\ \texttt{or}\ \cdots\ \texttt{or}\ e_n{:}s_n], ht, h, ch, mq) \to (P[a := s_i], ht, h, ch, mq)$
if $length(mq(a)) = 0 \wedge i \in [1..n] \wedge h(a) : ht \vdash e_i \to \texttt{true}$

// await with timeout clause, terminated by true condition
$(P[a \to \ell\ \texttt{await}\ e_1{:}s_1\ \texttt{or}\ \cdots\ \texttt{or}\ e_n{:}s_n\ \texttt{timeout}\ v{:}s], ht, h, ch, mq) \to (P[a := s_i], ht, h, ch, mq)$
if $length(mq(a)) = 0 \wedge i \in [1..n] \wedge h(a) : ht \vdash e_i \to \texttt{true}$

// await with timeout clause, terminated by timeout (occurs non-deterministically)
$(P[a \to \ell\ \texttt{await}\ e_1{:}s_1\ \texttt{or}\ \cdots\ \texttt{or}\ e_n{:}s_n\ \texttt{timeout}\ v{:}s], ht, h, ch, mq) \to (P[a := s], ht, h, ch, mq)$
if $length(mq(a)) = 0 \wedge h(a) : ht \vdash e_1 \to \texttt{false} \wedge \cdots \wedge h(a) : ht \vdash e_n \to \texttt{false}$

// context rule for expressions

$$\frac{h(a) : ht \vdash e \to e'}{(P[a \to C[e]], ht, h, ch, mq) \to (P[a := C[e']], ht, h, ch, mq)}$$

// context rule for statements

$$\frac{(P[a \to s], ht, h, ch, mq) \to (P[a := s'], ht', h', ch', mq')}{(P[a \to C[s]], ht, h, ch, mq) \to (P[a := C[s']], ht', h', ch', mq')}$$

Fig. 14. Transition relation for statements, Part 2.

$$
\begin{array}{l}
Init = \\
\{(P, ht, h, ch, mq) \in State \mid \\
\exists\, a_p \in ProcessAddress, \\
\quad a_r \in NonProcessAddress, \\
\quad a_s \in NonProcessAddress. \\
\quad a_r \neq a_s \\
\quad \wedge\, P = f_0[a_p := a_p.\texttt{main}()] \\
\quad \wedge\, ht = f_0[a_p := \texttt{process}, a_r := \texttt{sequence}, a_s := \texttt{sequence}] \\
\quad \wedge\, h = f_0[a_p := ha] \\
\quad \wedge\, ch = (\lambda(a_1, a_2) \in ProcessAddress \times ProcessAddress.\ \langle\rangle) \\
\quad \wedge\, mq = (\lambda a \in ProcessAddress.\ \langle\rangle) \\
\quad \text{where } ha = f_0[a_p := o_p, a_r := \langle\rangle, a_s := \langle\rangle] \\
\qquad\qquad o_p = f_0[\texttt{received} := a_r, \texttt{sent} := a_s]\}
\end{array}
$$

Fig. 15. Initial states.

REFERENCES

[1] Umut A. Acar, Guy E. Blelloch, and Robert Harper. 2006. Adaptive Functional Programming. *ACM Transactions on Programming Languages and Systems* 28, 6 (2006), 990–1034.

[2] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press.

[3] Frances E. Allen, John Cocke, and Ken Kennedy. 1981. Reduction of Operator Strength. In *Program Flow Analysis*, Steven S. Muchnick and Neil D. Jones (Eds.). Prentice-Hall, 79–101.

[4] P. Alvaro, T. Condie, N. Conway, J.M. Hellerstein, and R. Sears. 2010. I Do Declare: Consensus in a Logic Language. *ACM SIGOPS Operating Systems Review* 43, 4 (2010), 25–30.

[5] Gregory R. Andrews and Ronald A. Olsson. 1993. *The SR Programming Language: Concurrency in Practice.* Benjamin Cummings.

[6] Michael P. Ashley-Rollman, Peter Lee, Seth Copen Goldstein, Padmanabhan Pillai, and Jason D. Campbell. 2009. A Language for Large Ensembles of Independently Executing Nodes. In *Proceedings of the 25th International Conference on Logic Programming*. Springer, 265–280.

[7] Hagit Attiya and Jennifer Welch. 2004. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics* (2nd ed.). Wiley.

[8] Joshua S. Auerbach, Arthur P. Goldberg, Germán S. Goldszmidt, Ajei S. Gopal, Mark T. Kennedy, Josyula R. Rao, and James R. Russell. 1994. Concert/C: A Language for Distributed Programming. In *Proceedings of the USENIX Winter 1994 Technical Conference*. USENIX Association, Article 8, 18 pages.

[9] Antonio Badia. 2007. Question answering and database querying: Bridging the gap with generalized quantification. *Journal of Applied Logic* 5, 1 (2007), 3–19.

[10] Antonio Badia, Brandon Debes, and Bin Cao. 2008. An Implementation of a Query Language with Generalized Quantifiers. In *Proceedings of the 27th International Conference on Conceptual Modeling*. Springer, 547–548.

[11] Antonio Badia, Dirk Van Gucht, and Marc Gyssens. 1995. Query Languages with Generalized Quantifiers. In *Applications of Logic in Databases*, R. Ramakrishnan (Ed.). Kluwer Academic.

[12] Jason Baker, Chris Bond, James C. Corbett, J.J. Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Database Research*. 223–234.

[13] Berkeley Orders of Magnitude 2013. Bloom Programming Language. http://www.bloom-lang.net. (2013). Lastest release April 23, 2013. Accessed January 14, 2017.

[14] Mark Bickford. 2009. Component Specification Using Event Classes. In *Proceedings of the 12th International Symposium on Component-Based Software Engineering*. Springer, 140–155.

[15] Andrew P. Black, Norman C. Hutchinson, Eric Jul, and Henry M. Levy. 2007. The Development of the Emerald Programming Language. In *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages*. 11–1–11–51.

[16] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*. 335–350.

[17] Jiazhen Cai, Philippe Facon, Fritz Henglein, Robert Paige, and Edmond Schonberg. 1991. Type Analysis and Data Structure Selection. In *Constructing Programs from Specifications*, Bernhard Möller (Ed.). North-Holland, 126–164.

[18] Saksham Chand, Yanhong A. Liu, and Scott D. Stoller. 2016. Formal Verification of Multi-Paxos for Distributed Consensus. In *Proceedings of the 21st International Symposium on Formal Methods*. Springer, 119–136.

[19] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems* 26, 2 (2008), 4.

[20] Jens Claußen, Alfons Kemper, Guido Moerkotte, and Klaus Peithner. 1997. Optimizing Queries with Universal Quantification in Object-Oriented and Object-Relational Databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases*. Morgan Kaufman, 286–295.

[21] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). MIT Press.

[22] Flavio Cruz, Ricardo Rocha, Seth Copen Goldstein, and Frank Pfenning. 2014. A Linear Logic Programming Language for Concurrent Programming over Graph Structures. *Theory and Practice of Logic Programming* 14 (7 2014), 493–507.

[23] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[24] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 205–220.

[25] DistAlgo 2016. DistAlgo: A Language for Distributed Algorithms. http://github.com/DistAlgo. (2016). Beta release September 27, 2014. 1.0 release November 13, 2016.

[26] Erlang 2015. Erlang Programming Language. http://www.erlang.org/. (2015). Last released December 18, 2015.

[27] Colin J. Fidge. 1988. Timestamps in Message-Passing Systems That Preserve the Partial Ordering. In *Proceedings of the 11th Australian Computer Science Conference*. 56–66.

[28] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. 2011. Program transformation for development, verification, and synthesis of programs. *Intelligenza Artificiale* 5, 1 (2011), 119–125.

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

[30] Gautam and S. Rajopadhye. 2006. Simplifying Reductions. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 30–41.

[31] Chryssis Georgiou, Nancy A. Lynch, and Panayiotis Mavrommatis andJoshua A. Tauber. 2009. Automated Implementation of Complex Distributed Algorithms Specified in the IOA Language. *International Journal on Software Tools for Technology Transfer* 11, 2 (2009), 153–171.

[32] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. *ACM SIGOPS Operating Systems Review* 37, 5 (2003), 29–43.

[33] Michael Gorbovitski, Yanhong A. Liu, Scott D. Stoller, Tom Rothamel, and Tuncay Tekle. 2010. Alias Analysis for Optimization of Dynamic Languages. In *Proceedings of the 6th Symposium on Dynamic Languages*. ACM Press, 27–42.

[34] Deepak Goyal. 2000. *A Language Theoretic Approach to Algorithms*. Ph.D. Dissertation. Department of Computer Science, New York University.

[35] Adam Granicz, Daniel M. Zimmerman, and Jason Hickey. 2003. Rewriting UNITY. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications*. Springer, 138–147.

[36] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining Views Incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. 157–166.

[37] David Hansel, Rance Cleaveland, and Scott A. Smolka. 2004. Distributed prototyping from validated specifications. *Journal of Systems and Software* 70, 3 (2004), 275–298.

[38] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free coordination for Internet-scale Systems. In *USENIX Annual Technical Conference*. 9.

[39] Dilsun Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. 2010. *The Theory of Timed I/O Automata* (2nd ed.). Morgan & Claypool.

[40] Ingolf Heiko Krüger. 1996. *An experiment in compiler design for a concurrent object-based programming language*. Master's thesis. The University of Texas at Austin.

[41] A.D. Kshemkalyani and M. Singhal. 2008. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press.

[42] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[43] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21 (1978), 558–565. Issue 7.

[44] Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley.
[45] Leslie Lamport. 2009. The PlusCal Algorithm Language. In *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing*. Springer, 36–60.
[46] Jim Larson. 2009. Erlang for Concurrent Programming. *Commun. ACM* 52, 3 (2009), 48–56.
[47] Barbara Liskov. 1988. Distributed Programming in Argus. *Commun. ACM* 31, 3 (Mar. 1988), 300–312.
[48] Yanhong Annie Liu. 2013. *Systematic Program Design: From Clarity To Efficiency*. Cambridge University Press.
[49] Yanhong A. Liu, Jon Brandvein, Scott D. Stoller, and Bo Lin. 2016. Demand-Driven Incremental Object Queries. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*. ACM Press, 228–241.
[50] Yanhong A. Liu, Michael Gorbovitski, and Scott D. Stoller. 2009. A Language and Framework for Invariant-Driven Transformations. In *Proceedings of the 8th International Conference on Generative Programming and Component Engineering*. ACM Press, 55–64.
[51] Yanhong A. Liu and Scott D. Stoller. 2003. Dynamic Programming via Static Incrementalization. *Higher-Order and Symbolic Computation* 16, 1–2 (2003), 37–62.
[52] Yanhong A. Liu and Scott D. Stoller. 2009. From Datalog Rules to Efficient Programs with Time and Space Guarantees. *ACM Transactions on Programming Languages and Systems* 31, 6 (2009), 1–38.
[53] Yanhong A. Liu, Scott D. Stoller, Michael Gorbovitski, Tom Rothamel, and Yanni E. Liu. 2005. Incrementalization Across Object Abstraction. In *Proceedings of the 20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 473–486.
[54] Yanhong A. Liu, Scott D. Stoller, Ning Li, and Tom Rothamel. 2005. Optimizing Aggregate Array Computations in Loops. *ACM Transactions on Programming Languages and Systems* 27, 1 (2005), 91–125.
[55] Yanhong A. Liu, Scott D. Stoller, and Bo Lin. 2012. High-Level Executable Specifications of Distributed Algorithms. In *Proceedings of the 14th International Symposium on Stabilization, Safety, and Security of Distributed Systems*. Springer, 95–110.
[56] Yanhong A. Liu, Scott D. Stoller, Bo Lin, and Michael Gorbovitski. 2012. From Clarity to Efficiency for Distributed Algorithms. In *Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*. 395–410.
[57] Yanhong A. Liu, Chen Wang, Michael Gorbovitski, Tom Rothamel, Yongxi Cheng, Yingchao Zhao, and Jing Zhang. 2006. Core Role-Based Access Control: Efficient Implementations by Transformations. In *Proceedings of the ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation*. 112–120.
[58] Nuno P. Lopes, Juan A. Navarro, Andrey Rybalchenko, and Atul Singh. 2010. Applying Prolog to Develop Distributed Systems. *Theory and Practice of Logic Programming* 10, 4-6 (July 2010), 691–707.
[59] Nancy A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufman.
[60] Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*. North-Holland, 120–131.
[61] Petar Maymounkov and David Mazières. 2002. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Peer-to-Peer Systems*. 53–65.
[62] MPI Last released June 4, 2015. Message Passing Interface Forum. http://www.mpi-forum.org/. (Last released June 4, 2015).
[63] Hiroaki Nakamura. 2001. Incremental Computation of Complex Object Queries. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 156–165.
[64] Robert Paige. 1989. Real-Time Simulation of a Set Machine on a RAM. In *Proceedings of the International Conference on Computing and Information*. Canadian Scholars Press, 69–73.
[65] Robert Paige and Shaye Koenig. 1982. Finite Differencing of Computable Expressions. *ACM Transactions on Programming Languages and Systems* 4, 3 (1982), 402–454.
[66] Vyacheslav Petukhin. 1997. Programs with Universally Quantified Embedded Implications. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer, 310–324.
[67] PRL Project. 2013. EventML. http://www.nuprl.org/software/#WhatisEventML. (2013). Lastest release September 21, 2012. Accessed January 14, 2017.
[68] William Pugh and Tim Teitelbaum. 1989. Incremental Computation via Function Caching. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*. 315–328.
[69] G. Ramalingam and Thomas Reps. 1993. A Categorized Bibliography on Incremental Computation. In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages*. 502–510.
[70] Michel Raynal. 1988. *Distributed Algorithms and Protocols*. Wiley.

[71] Michel Raynal. 2010. *Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems*. Morgan & Claypool.
[72] Michel Raynal. 2013. *Distributed Algorithms for Message-Passing Systems*. Springer.
[73] Tom Rothamel and Yanhong A. Liu. 2008. Generating Incremental Implementations of Object-Set Queries. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*. ACM Press, 55–66.
[74] Antony Rowstron and Peter Druschel. 2001. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*. Springer, 329–350.
[75] Diptikalyan Saha and C. R. Ramakrishnan. 2003. Incremental Evaluation of Tabled Logic Programs. In *Proceedings of the 19th International Conference on Logic Programming*. Springer, 392–406.
[76] Michael L Scott. 1991. The Lynx Distributed Programming Language: Motivation, Design, and Experience. *Computer Languages* 16, 3 (1991), 209–233.
[77] Traian Florin Serbanuta, Grigore Rosu, and Jose Meseguer. 2009. A Rewriting Logic Approach to Operational Semantics. *Information and Computation* 207 (2009), 305–340. Issue 2.
[78] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*. IEEE CS Press, 1–10.
[79] I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan. 2003. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on Networking* 11, 1 (2003), 17–32.
[80] Theresa Swift, David S. Warren, and others. 2016. *The XSB System Version 3.7,x*. http://xsb.sourceforge.net. Latest release July 6, 2016.
[81] Gerard Tel. 2000. *Introduction to Distributed Algorithms* (2nd ed.). Cambridge University Press.
[82] Robbert van Renesse and Deniz Altinbuken. 2015. Paxos Made Moderately Complex. *Comput. Surveys* 47, 3 (Feb. 2015), 42:1–42:36.
[83] Robbert van Renesse and Fred B. Schneider. 2004. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 91–104.
[84] Dan E. Willard. 1984. Efficient Processing of Relational Calculus Expressions Using Range Query Theory. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. 164–175.
[85] Dan E. Willard. 2002. An Algorithm for Handling Many Relational Calculus Queries Efficiently. *J. Comput. System Sci.* 65 (2002), 295–331.
[86] Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115 (1994), 38–94.
[87] Ben Y Zhao, Ling Huang, Jeremy Stribling, Sean C Rhea, Anthony D Joseph, and John D Kubiatowicz. 2004. Tapestry: A Resilient Global-Scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications* 22, 1 (2004), 41–53.

Received January 2015; revised December 2015; accepted September 2016