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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2017 ACM. 0164-0925/2017/1-ART1 $15.00 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 quantification 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: $(a,b) < (a_2,b_2)$ iff $a < a_2$ or $a = a_2$ and $b < b_2$) 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 $T_m;P_i \text{ requests resource}$ 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 $T_m;P_i \text{ requests resource}$, 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 $T_m;P_i \text{ requests resource}$ message from its request queue and sends a (timestamped) $P_i \text{ releases resource}$ message to every other process.

4. When process $P_j$ receives a $P_i \text{ releases resource}$ message, it removes any $T_m;P_i \text{ requests resource}$ message from its request queue.

5. Process $P_i$ is granted the resource when the following two conditions are satisfied:
   (i) There is a $T_m;P_i \text{ requests resource}$ 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.

\[
\text{class } p \text{ extends process:}
\[
\text{process_body}
\]

A special method \( \text{setup} \) may be defined in process_body for initially setting up data in the process before the process’s execution starts. A special method \( \text{run()} \) may be defined in process_body for carrying out the main flow of execution. A special variable \( \text{self} \) 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 \( \text{node}_\text{exp} \), 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.

\[
n \text{new} \ p \ \text{at} \ \text{node}_\text{exp}
\]

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 \( \text{setup} \) method. A call \( \text{start()} \) on the process then starts the execution of its \( \text{run()} \) method.

A statement for sending messages is of form (3) below. It sends the message that is the value of expression \( m\text{exp} \) to the process or set of processes that is the value of expression \( p\text{exp} \).

\[
\text{send} \ m\text{exp} \ \text{to} \ p\text{exp}
\]

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.

\[
-- \ l
\]

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 \( l_1, \ldots, l_j \), 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.

\[
\begin{align*}
\text{receive } mexp_1 \text{ from } pexp_1, \ldots, mexp_k \text{ from } pexp_k \\
\text{at } l_1, \ldots, l_j: \\
\text{handler_body}
\end{align*}
\]

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.

\[
\begin{align*}
\text{await } bexp \\
\text{await timeout } t
\end{align*}
\]

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

\[
\begin{align*}
\text{await } bexp_1: stmt_1 \\
\text{or } \ldots \\
\text{or } bexp_k: stmt_k \\
\text{timeout } t: stmt
\end{align*}
\]

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 \( \forall v_1 \in \text{sexp}_1, \ldots, v_k \in \text{sexp}_k \mid bexp \), 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 \( bexp \).

\[
\begin{align*}
\text{some } & v_1 \in \text{sexp}_1, \ldots, v_k \in \text{sexp}_k \mid bexp \\
\text{each } & v_1 \in \text{sexp}_1, \ldots, v_k \in \text{sexp}_k \mid bexp
\end{align*}
\]  

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

\[
\text{each } x \in s, x2 \in s2 \mid x > x2
\]

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.

\[
\text{some } x \in s \mid \text{each } x2 \in s2 \mid x > x2
\]

- 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 \( \forall v_1 \in \text{sexp}_1 \mid bexp \).

\[
\{ \text{exp: } v_1 \in \text{sexp}_1, \ldots, v_k \in \text{sexp}_k \mid bexp \}\]

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

\[
\{x*y: x \in s, x2 \in s2 \mid x > x2\}
\]

We abbreviate \( \{ v: v \in \text{sexp} \mid bexp \} \) as \( \{ v \in \text{sexp} \mid bexp \} \).

- 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 \( \text{count, sum, min, and max} \). 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 \( \forall v_1 \in \text{sexp}_i \mid bexp \), for the second form.

\[
\begin{align*}
\text{agg } & \text{sexp} \\
\text{agg} \{ \text{exp: } v_1 \in \text{sexp}_1, \ldots, v_k \in \text{sexp}_k \mid bexp \}
\end{align*}
\]

ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, Article 1. Publication date: January 2017.
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 $\emptyset$ 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 $n \text{ new } P$, 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)) #
10        # wait for own req < others in q
11        # and for acks from all in s
12        await each ('request', t2, p2) in q | # 5 in Fig 1
13           (t2,p2) != (t,self) implies (t,self) < (t2,p2)
14        and each p2 in s | #
15           some received('ack', t2, =p2) | t2 > t
16        task() # critical section
17     -- release
18         q.del(('request', t, self)) # 3 in Fig 1
19         send ('release', logical_time(), self) to s #
20     receive ('request', t2, p2): # 2 in Fig 1
21         q.add(('request', t2, p2)) #
22         send ('ack', logical_time(), self) to p2 #
23     receive ('release', _, p2): # 4 in Fig 1
24         for ('request', t2, =p2) in q: #
25             q.del(('request', t2, p2)) #
26     def run(): # main method for the process
27         ... # do non-CS tasks of the process
28     def task(): ... # define critical section task
29     mutex(task) # run task with mutual exclusion
30         ... # do non-CS tasks of the process
31     def main(): # main method for the application
32         ... # do other tasks of the application
33     configure channel = {reliable, fifo} # use reliable and FIFO channel
34     configure clock = Lamport # use Lamport clock
35     ps := 50 new P # create 50 processes of P class
36     for p in ps: p.setup(ps-{p}) # pass to each process other processes
37     for p in ps: p.start() # start the run method of each process
38         ... # 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.
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,

\[
\text{each } x \text{ in } s | \text{some } x_2 \text{ in } s_2 | \text{bexp}
\]
would be converted using rule 1 to
\[
each x \text{ in } s \mid \text{count } \{x2 \text{ in } s2 \mid \text{bexp}\} \neq 0
\]
and then using rule 2 to
\[
\text{count } \{x \text{ in } s \mid \text{count } \{x2 \text{ in } s2 \mid \text{bexp}\} \neq 0\} = \text{count } s
\]
A simpler conversion is possible for this example, using a rule in Table 2, described next.

Table 1. Rules for converting single quantifications.

<table>
<thead>
<tr>
<th>Quantification</th>
<th>Aggregation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 some x in s</td>
<td>count {x in s</td>
</tr>
<tr>
<td>2 each x in s</td>
<td>count {x in s</td>
</tr>
<tr>
<td>3</td>
<td>count {x in s</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Nested Quantifications</th>
<th>Aggregation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 some x in s</td>
<td>some x2 in s2</td>
</tr>
<tr>
<td>2 each x in s</td>
<td>some x2 in s2</td>
</tr>
<tr>
<td>3 some x in s</td>
<td>each x2 in s2</td>
</tr>
<tr>
<td>4 each x in s</td>
<td>each x2 in s2</td>
</tr>
<tr>
<td>5</td>
<td>count {(x,x2): x in s, x2 in s2 \mid not bexp} = 0</td>
</tr>
</tbody>
</table>

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,
\[
each x \text{ in } s \mid \text{bexp implies } y < x
\]
can be converted using rule 8 in Table 4 to
Table 3. Rules for single quantified order comparison.

<table>
<thead>
<tr>
<th>Existential Aggregation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 some x in s</td>
</tr>
<tr>
<td>2 some x in s</td>
</tr>
<tr>
<td>3 some x in s</td>
</tr>
<tr>
<td>4 some x in s</td>
</tr>
<tr>
<td>5 some x in s</td>
</tr>
<tr>
<td>6 some x in s</td>
</tr>
<tr>
<td>7 some x in s</td>
</tr>
<tr>
<td>8 some x in s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Universal Aggregation</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 each x in s</td>
</tr>
<tr>
<td>10 each x in s</td>
</tr>
<tr>
<td>11 each x in s</td>
</tr>
<tr>
<td>12 each x in s</td>
</tr>
<tr>
<td>13 each x in s</td>
</tr>
<tr>
<td>14 each x in s</td>
</tr>
<tr>
<td>15 each x in s</td>
</tr>
<tr>
<td>16 each x in s</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Quantification</th>
<th>Decomposed Quantifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 some x in s</td>
<td>not e not each x in s</td>
</tr>
<tr>
<td>2 some x in s</td>
<td>e1 and e2 some x in {x in s</td>
</tr>
<tr>
<td>3 some x in s</td>
<td>e1 or e2 (some x in s</td>
</tr>
<tr>
<td>4 some x in s</td>
<td>e1 implies e2 (some x in s</td>
</tr>
<tr>
<td>5 each x in s</td>
<td>not e not some x in s</td>
</tr>
<tr>
<td>6 each x in s</td>
<td>e1 and e2 (each x in s</td>
</tr>
<tr>
<td>7 each x in s</td>
<td>e1 or e2 each x in {x in s</td>
</tr>
<tr>
<td>8 each x in s</td>
<td>e1 implies e2 each x in {x in s</td>
</tr>
</tbody>
</table>

**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>
<thead>
<tr>
<th>Query</th>
<th>Replacement</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>count s</td>
<td>number</td>
<td>O(1)</td>
</tr>
<tr>
<td>Updates</td>
<td>Inserted</td>
<td></td>
</tr>
<tr>
<td>s := {}</td>
<td>number := 0</td>
<td>O(1)</td>
</tr>
<tr>
<td>s.add(x)</td>
<td>if x not in s: number += 1</td>
<td>O(1)</td>
</tr>
<tr>
<td>s.del(x)</td>
<td>if x in s: number -= 1</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query</th>
<th>Replacement</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>max s</td>
<td>maximum</td>
<td>O(1)</td>
</tr>
<tr>
<td>Updates</td>
<td>Inserted</td>
<td></td>
</tr>
<tr>
<td>s := {x}</td>
<td>maximum := x</td>
<td>O(1)</td>
</tr>
<tr>
<td>s.add(x)</td>
<td>if x &gt; maximum: maximum := x</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query</th>
<th>Replacement</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>max s</td>
<td>ds.max()</td>
<td>O(1)</td>
</tr>
<tr>
<td>Updates</td>
<td>Inserted</td>
<td></td>
</tr>
<tr>
<td>s := {}</td>
<td>ds := new DS()</td>
<td>O(1)</td>
</tr>
<tr>
<td>s := {x}</td>
<td>ds := new DS(); ds.add(x)</td>
<td>O(1)</td>
</tr>
<tr>
<td>s.add(x)</td>
<td>if x not in s: ds.add(x)</td>
<td>O(log</td>
</tr>
<tr>
<td>s.del(x)</td>
<td>if x in s: ds.del(x)</td>
<td>O(log</td>
</tr>
</tbody>
</table>

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 \( \{ x \in s \mid bexp \} \) 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 \( (\text{\textquotesingle}ack\text{\textquotesingle},t2,=p2) \) in `received` is used in place of \( \text{\textquotesingle}received\text{\textquotesingle}(\text{\textquotesingle}ack\text{\textquotesingle},t2,=p2) \).

\[
\text{each} \ (\text{\textquotesingle}request\text{\textquotesingle}, \ t2, \ p2) \ \text{in} \ q \ |
  (t2,p2) \neq (t,self) \ \text{implies} \ (t,self) < (t2,p2)
\]

and each \( p2 \ \text{in} \ s \ |
\text{some} \ (\text{\textquotesingle}ack\text{\textquotesingle}, \ t2, =p2) \ \text{in} \ \text{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:

\[
\text{count} \ \{(\text{\textquotesingle}request\text{\textquotesingle}, \ t2, \ p2) \ \text{in} \ q \mid
  (t,self) > (t2,p2)\} = 0
\]

and

\[
\text{count} \ \{p2: \ p2 \ \text{in} \ s, \ (\text{\textquotesingle}ack\text{\textquotesingle}, \ t2, \ p2) \ \text{in} \ \text{received} \mid \ t2 > t\} = \text{count} \ s
\]

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

  Note that when \( (\text{\textquotesingle}request\text{\textquotesingle},t,self) \) in particular is added to or removed from \( q \), \( \text{earlier} \) and \( \text{number1} \) 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 \( \text{responded} \), \( \text{number2} \), and \( \text{total} \), respectively, so the conjunct becomes \( \text{number2} = \text{total} \); when \( s \) is initialized in \( \text{setup} \), assign \( \text{total} \) the size of \( s \), taking \( O(|s|) \) time, done only once...
for each process; when \( t \) is assigned a new value, let \( \text{responded} \) be \( \{\} \), and let \( \text{number2} \) be 0, taking \( O(1) \) time; when an \( \text{ack} \) message is added to \( \text{received} \), if the associated conditions hold, increment \( \text{number2} \) by 1, taking \( O(1) \) time. A test of definedness of \( t \) is omitted in the maintenance for receiving \( \text{ack} \) 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 \( \text{logical\_time()} \) returns a timestamp larger than all existing timestamp values, and thus at the assignment to \( t \) in method \( \text{mutex} \), we have that \( \text{earlier} \) is \( q \) and \( \text{responded} \) is \( \{\} \). So, an incrementalization rule for maintaining \( \text{earlier} \) specifies that at update \( t := \text{logical\_time()} \), the maintenance is \( \text{earlier} := q \); similarly for maintaining \( \text{responded} \). 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 \( \text{await} \) 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 \( \text{ack} \) messages. Using based representation for sets [17, 34, 64], maintaining \( \text{earlier} \) and \( \text{responded} \) can each be done using one bit for each process.

Note that the sequence \( \text{received} \) 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: \( \text{earlier} \), \( \text{number1} \), \( \text{responded} \), \( \text{number2} \), and \( \text{total} \), a drastic space improvement from unbounded for \( \text{received} \) to linear in the number of processes.

**Example with naive use of aggregation operator \( \text{min} \).** 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 \( \text{min} \) 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 \( \text{min} \) to be correct, special care must be taken to deal with the case when the argument to \( \text{min} \) is empty, because then \( \text{min} \) is undefined.

Consider the first conjunct in the synchronization condition in the \( \text{await} \) statement in Figure 2, copied below:

\[
\begin{align*}
\text{each ('request', t2, p2) in q |} \\
(t2,p2) != (t,self) \text{ implies (t,self) < (t2,p2)}
\end{align*}
\]

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

\[
(t,self) < \text{min} \{(t2,p2): ('request', t2, p2) in q} \\
| (t2,p2) != (t,self)
\]

However, that is incorrect, because the argument of \( \text{min} \) may be empty, in which case \( \text{min} \) is undefined.

Instead of resorting to commonly used special values, such as \( \text{maxint} \), 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))
                          break
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 \texttt{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 \texttt{earlier} and \texttt{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 \texttt{received} removed, just as in Figure 3. The difference is that maintaining ds for incrementalizing \texttt{min} under element addition to and deletion from q takes \(O(\log |s|)\) time, as opposed to \(O(1)\) time for maintaining \texttt{earlier} and \texttt{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 \texttt{receive} definition for \texttt{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
class P extends process:
    def setup(s):
        self.s := s
        self.total := count s  # total num of other processes
        self.q := {}
        self.ds := new DS()  # data structure for maintaining
                            # requests by other processes
        def mutex(task):
            -- request
            self.t := logical_time()
            self.responded := {}  # set of responded processes
            self.number := 0  # num of responded processes
            send ('request', t, self) to s
            q.add(('request', t, self))
            await (ds.is_empty() or (t, self) < ds.min())
            and number = total  # use maintained results
            task()
            -- release
            q.del(('request', t, self))
            send ('release', logical_time(), self) to s
        receive ('request', t2, p2):
            ds.add((t2, p2))  # add to data structure
            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
                        number += 1  # increment number
        receive ('release', _, p2):
            for ('request', t2, =p2) in q:
                ds.del((t2, p2))  # delete from data structure
                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.
1 class P extends process:
2   def setup(s):
3     self.s := s
4
5   def mutex(task):
6     -- request
7     self.t := logical_time()
8     send ('request', t, self) to s
9     await each received ('request', t2, p2) |
10        not (some received ('release', t3, =p2) | t3 > t2)
11        implies (t, self) < (t2, p2)
12     and each p2 in s |
13        some received ('ack', t2, =p2) | t2 > t
14     task()
15     -- release
16     send ('release', logical_time(), self) to s
17
18   receive ('request', _, p2):
19     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 to 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 lightweight 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

\[
\text{await count \{a: received ('p2b',=b) from a\} > (count acceptors)/2}
\]

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 facilitates 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 $\theta$ 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 $\ell$, replacing the empty label in that await statement with $\ell$, and inserting $\ell$ 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)
            instanceof(Expression, ClassName)
            and(Expression, Expression)   // conjunction (short-circuiting)
            or(Expression, Expression)   // disjunction (short-circuiting)
            each Iterator | Expression
            some Iterator | Expression

Tuple ::= (Expression*)

(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 \(s \mid b\). Let \(x\) be a fresh
      variable. Let \(\theta\) 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

ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, Article 1. Publication date: January 2017.
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 \(\tilde{e}_j\) denote \(e_j\) after removing the "=" prefix, if any. The quantification is rewritten as some \(x\) in \(s\) \mid isTuple(x) and len(x) is \(n\) and (select(x, \(j_1\)), ..., select(x, \(j_m\))) is \((\tilde{e}_{i_1}, \ldots, \tilde{e}_{i_k})\) 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 \(\theta\) 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 \(\tilde{e}_j\) 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.

```python
S := e
if isinstance(S, set):
    S := { x : x in S | isTuple(x) and len(x) is n
    and (select(x,j_1), ..., select(x,j_m)) is (\bar{e}_{j_1}, ..., \bar{e}_{j_m}) }
for x in S:
    s\theta
else: // S is a sequence
    for x in S:
        if (isTuple(x) and len(x) is n
        and (select(x,j_1), ..., select(x,j_m)) is (\bar{e}_{j_1}, ..., \bar{e}_{j_m}):
        s\theta
        else:
            skip
```

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

(7) Comprehensions in which some variables are prefixed with \(=\) are translated into com-
prehensions 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
\[ x := \{ e | x_1 \text{ in } e_1, \ldots, x_n \text{ in } e_n | b \} \]
where each \(x_i\) is a pattern, is replaced with
\[ x := \text{new set for } x_1 \text{ in } e_1: \]
\[ \ldots \]
\[ \text{for } x_n \text{ in } e_n: \]
\[ \text{if } b: \]
\[ x.\text{add}(e) \]

(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 \(\ell\) on a statement
\(s\) other than **await** is treated as syntactic sugar for label \(\ell\) on **await** \(\text{true} : \text{skip}\) 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 \(D\).
- \(\text{Set}(D)\) contains finite sets of values from domain \(D\).
- \(D_1 \rightarrow D_2\) contains partial functions from \(D_1\) to \(D_2\). \(\text{dom}(f)\) is the domain of a partial function \(f\).

Notes.

- We require that ProcessAddress and NonProcessAddress be disjoint.
- For \(a \in \text{ProcessAddress}\) and \(h \in \text{Heap}\), \(h(a)\) is the local heap of process \(a\). For \(a \in \text{Address}\) and \(ht \in \text{HeapType}\), \(ht(a)\) 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.
\[
\begin{align*}
\text{Bool} & = \{\text{true}, \text{false}\} \\
\text{Int} & = \ldots \\
\text{ProcessAddress} & = \ldots \\
\text{NonProcessAddress} & = \ldots \\
\text{Address} & = \text{ProcessAddress} \cup \text{NonProcessAddress} \\
\text{Tuple} & = \text{Val}^* \\
\text{Val} & = \text{Bool} \cup \text{Int} \cup \text{Address} \cup \text{Tuple} \\
\text{SetOfVal} & = \text{Set}(\text{Val}) \\
\text{SeqOfVal} & = \text{Val}^* \\
\text{Object} & = (\text{Field} \rightarrow \text{Val}) \cup \text{SetOfVal} \cup \text{SeqOfVal} \\
\text{HeapType} & = \text{Address} \rightarrow \text{Name} \\
\text{LocalHeap} & = \text{Address} \rightarrow \text{Object} \\
\text{Heap} & = \text{ProcessAddress} \rightarrow \text{LocalHeap} \\
\text{ChannelStates} & = \text{ProcessAddress} \times \text{ProcessAddress} \rightarrow \text{Tuple}^* \\
\text{MsgQueue} & = (\text{Tuple} \times \text{ProcessAddress})^* \\
\text{State} & = (\text{ProcessAddress} \rightarrow \text{Statement}) \times \text{HeapType} \times \text{Heap} \times \text{ChannelStates} \\
& \times (\text{ProcessAddress} \rightarrow \text{MsgQueue})
\end{align*}
\]

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.

\[
\begin{align*}
\text{Expression} & ::= \text{Address} \\
& \quad \text{Address.Field} \\
\text{Statement} & ::= \text{for Variable intuple Tuple: Statement}
\end{align*}
\]

Fig. 10. Extensions to the abstract syntax.

The statement \( \text{for } v \text{ intuple } t: s \) 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.
\[
\begin{array}{l}
\text{Val ::= Literal} \\
\text{Address} \\
(\text{Val}^*) \\
C ::= [] \\
(\text{Val}^*, \text{Expression}^*) \\
\text{C.MethodName(Expressio}n^*) \\
\text{Address.MethodName(Val}^*, \text{Expression}^*) \\
\text{UnaryOp(C)} \\
\text{BinaryOp(C, Expression)} \\
\text{BinaryOp(Val, C)} \\
isinstance(C, \text{ClassName}) \\
or(C, \text{Expression}) \\
some \text{Pattern in } C \mid \text{Expression} \\
\text{C.Field ::= Expression} \\
\text{Address.Field ::= } C \\
\text{InstanceVariable ::= } C \\
\text{C ; Statement} \\
\text{if } C : \text{Statement else: Statement} \\
\text{for InstanceVariable in } C : \text{Statement} \\
\text{for InstanceVariable intuple Tuple: } C \\
send C \text{ to Expression} \\
send Val \text{ to } C \\
\text{await Expression : Statement AnotherAwaitClause}^* \text{ timeout } C
\end{array}
\]

Fig. 11. Evaluation contexts.

A.5 Transition Relations

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

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 \(\text{Val}\)); and \(\ell\) 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 \(f[x \rightarrow y]\) iff \(f(x)\) equals \(y\). For example, in transition rules for statements, a function \(P\) in \(\text{ProcessAddress} \rightarrow \text{Statement}\) matches \(P[a \rightarrow s]\) 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 \(f\), \(f \oplus a\) 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., \(\langle 0, 1, 2 \rangle \in \text{Int}^*\).
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(c₁, c₂) holds iff class c₁ is a descendant of class c₂ in the inheritance hierarchy.
• For c ∈ ClassName, new(c) returns a new instance of c.

\[
\text{new}(c) = \begin{cases} 
\{\} & \text{if } c = \text{set} \\
\{\} & \text{if } c = \text{sequence} \\
\bot & \text{otherwise} 
\end{cases}
\]

• For m ∈ MethodName and c ∈ 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, h̅, h̅′ ∈ LocalHeap and ht, ht′ ∈ HeapType and v, ũ ∈ Val, the relation isCopy(v, h, h̅, ht, ũ, h̅′, 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) ũ is a copy of v for a process whose local heap was h̅ before v was copied into it and whose local heap is h̅′ after v is copied into it, i.e., ũ is the same as v except that, instead of referencing objects in h, it references newly created copies of those objects in h̅′, and (3) h̅′ and ht′ are versions of h̅ 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 ũ, and new copies of process objects are not created. We give auxiliary definitions and then a formal definition of isCopy.

For v ∈ Val, let addrsv(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,

\[
a ∈ \text{addrsv}(v, h) ⇔ \\
(v ∈ \text{Address} ∧ v = a) ∨ (v ∈ \text{dom}(h) ∧ h(v) ∈ \text{Field} → \text{Val} ∧ (∃f ∈ \text{dom}(h(v)). a ∈ \text{addrsv}(h(v)(f), h))) ∨ (v ∈ \text{dom}(h) ∧ h(v) ∈ \text{SetOfVal} ∪ \text{SeqOfVal} ∧ (∃v′ ∈ h(v). a ∈ \text{addrsv}(v′, h))) ∨ (∃v₁, . . . , vₙ ∈ \text{Val}. v = (v₁, . . . , vₙ) ∧ ∃i ∈ [1..n]. a ∈ \text{addrsv}(v₁, h))
\]

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

\[
\text{subst}(v, ũ, f) ⇔ (v ∈ \text{Bool} ∪ \text{Int} ∪ (\text{Address} \setminus \text{dom}(f)) ∧ ũ = v) ∨ (v ∈ \text{dom}(f) ∧ f(ũ) = v) ∨ (∃v₁, . . . , vₙ, ũ₁, . . . , ũₙ. v = (v₁, . . . , vₙ) ∧ ũ = (ũ₁, . . . , ũₙ) ∧ (∀i ∈ [1..n]. \text{subst}(vᵢ, ũᵢ, f)))
\]

Similarly, for o, ō ∈ Object and f ∈ Address → Address, the relation subst(o, ō, f) holds iff o is obtained from ō 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):

\[
\text{isCopy}(v, h, \tilde{h}, \tilde{h}', h') \Leftrightarrow \\
\exists A \subseteq \text{NonProcessAddress}. \exists f \in A^{1-1} (\text{addr}(v, h) \setminus \text{ProcessAddress}). \\
A \cap \text{dom}(h) = \emptyset \land \text{dom}(h') = \text{dom}(h) \cup A \land \text{dom}(\tilde{h}) = \text{dom}(h) \cup A \\
\land (\forall a \in \text{dom}(h). h'(a) = h(a)) \land (\forall a \in \text{dom}(\tilde{h}). \tilde{h}'(a) = h(a)) \\
\land (\forall a \in A. \text{ht}(a) = h(f(a)) \land \text{subst}(h(a), \tilde{h}'(a), f))
\]

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

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

If \(\text{matchesDefLbl}(d, \ell) \land \text{findSubst}(m, a, h, d) \neq \bot,\) then \(\text{matchRcvDef}(m, a, \ell, h, d)\) returns \(s\theta,\) where \(s\) is the body of \(d\) (i.e., the statement that appears in \(d\)) and \(\theta = \text{findSubst}(m, a, h, d),\) otherwise it returns \(\bot.\)

• For \(m \in \text{Val}, a \in \text{ProcessAddress}, \ell \in \text{Label}, c \in \text{ClassName},\) and \(h \in \text{LocalHeap},\) if message \(m\) can be received from \(a\) at label \(\ell\) in class \(c\) by a process with local heap \(h,\) then \(\text{receiveAtLabel}((m, a), \ell, c, h)\) 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 \(\text{matchRcvDef}(m, a, \ell, h, d)\) is not \(\bot,\) then, letting \(d_1, \ldots, d_n\) be the receive definitions in \(c\) such that \(\text{matchRcvDef}(m, a, \ell, h, d_i)\) is not \(\bot,\) and letting \(s_i = \text{matchRcvDef}(m, a, \ell, h, d_i), \text{receiveAtLabel}((m, a), \ell, c, h)\) returns \(\{s_1, \ldots, s_n\}\). Otherwise, \(\text{receiveAtLabel}((m, a), \ell, c, h)\) returns \(\emptyset.\)

### A.6 Executions

An execution is a sequence of transitions \(\sigma_0 \to \sigma_1 \to \sigma_2 \to \cdots\) such that \(\sigma_0\) 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 ⊩ a.f → h(a)(f) if a ∈ dom(h) ∧ f ∈ dom(h(a))

// invoke method in user-defined class
ht : h ⊩ a.m(v₁, . . . , vₙ) → e[self := a, x₁ := v₁, . . . , xₙ := vₙ ]
   if a ∈ dom(h) ∧ methodDef (ht(a), m, defun m(x₁, . . . , xₙ) e)

// invoke method in pre-defined class (representative examples)
ht : h ⊩ a.contains(v₁) → true if a ∈ dom(h) ∧ ht(a) = set ∧ v₁ ∈ h(a)
ht : h ⊩ a.contains(v₁) → false if a ∈ dom(h) ∧ ht(a) = set ∧ v₁ ∈ h(a)

// unary operations
ht : h ⊩ not(true) → false
ht : h ⊩ not(false) → true
ht : h ⊩ istuple(v) → true if v is a tuple
ht : h ⊩ istuple(v) → false if v is not a tuple
ht : h ⊩ len(v) → n if v is a tuple with n components

// binary operations
ht : h ⊩ plus(v₁, v₂) → v₃ if v₁ ∈ Int ∧ v₂ ∈ Int ∧ v₃ = v₁ + v₂
   if v₂ ∈ Int ∧ v₂ > 0 ∧ (v₁ is a tuple with at least v₂ components)
   ∧ (v₃ is the v₂ th component of v₁)

// instanceof
ht : h ⊩ instanceof(a, c) → true if ht(a) = c
ht : h ⊩ instanceof(a, c) → false if ht(a) ≠ c

// disjunction
ht : h ⊩ or(true, e) → true
ht : h ⊩ or(false, e) → e

// existential quantification
ht : h ⊩ some x in a | e → e[x := v₁] or . . . or e[x := vₙ ]
   if (ht(a) = sequence ∧ h(a) = (v₁, . . . , vₙ ))
   ∨ (ht(a) = set ∧ (v₁, . . . , vₙ )) 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 → a'.f := v], ht, h[a := ha[a' → o]], ch, mq)
→ (P[a := skip], ht, h[a := ha[′ := o[f := v]]], ch, mq)

// object creation
(P[a → a'.f := new c], ht, h[a := ha[a' → o]], ch, mq)
→ (P[a := skip], ht[a' := c], h[a := ha[′ := o[f := a.c]], a.c := new(c)], ch, mq)
if a.c ∈ dom(ht) ∧ a.c ∈ Address ∧ (a.c ∈ ProcessAddress ⇔ extends(c, process))

// sequential composition
(P[a → skip; s], ht, h, ch, mq) → (P[a := s], ht, h, ch, mq)

// conditional statement
(P[a → if true : s₁ else : s₂], ht, h, ch, mq) → (P[a := s₁], ht, h, ch, mq)

(P[a → if false : s₁ else : s₂], ht, h, ch, mq) → (P[a := s₂], ht, h, ch, mq)

// for loop
(P[a → for x in a'.s], ht, h, ch, mq) → (P[a := for x intuple (v₁, . . . , vₙ) : s], ht, h, ch, mq)
if((ht(a) = sequence ∧ h(a)(a′)) = (v₁, . . . , vₙ))
∨ (ht(a) = set ∧ (v₁, . . . , vₙ) is a linearization of h(a)(a′))

(P[a → for x intuple (v₁, . . . , vₙ) : s], ht, h, ch, mq)
→ (P[a := s[x := v₁]; for x intuple (v₂, . . . , vₙ) : s], ht, h, ch, mq)

(P[a → for x intuple () : s], ht, h, ch, mq) → (P[a := skip], ht, h, ch, mq)

// while loop
(P[a → while e : s], ht, h, ch, mq) → (P[a := if e : (s; while e : s) else : skip], ht, h, ch, mq)

// invoke method in user-defined class
(P[a → a'.m(v₁, . . . , vₙ)], ht, h, ch, mq)
→ (P[a := s[self := a, x₁ := v₁, . . . , xₙ := vₙ]], ht, h, ch, mq)
if a′ ∈ dom(h(a))
∧ h(a′) ∉ {process, set, sequence} ∧ methodDef(h(a′), m, def m(x₁, . . . , xₙ) 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 → a'.start()], ht, h[a → ha[a' → o]], ch, mq)
→ (P[a := skip, a' := a'.run()], ht[aₛ := sequence, aᵣ := sequence],
h[a := ha ⊗ a', a' := fₒ[a' → o]sent := aₛ, received := aᵣ], aₛ := ⟨⟩, aᵣ := ⟨⟩], ch, mq)
if extends(ht(a'), process) ∧ (ht(a′) inherits start from process) ∧ aᵣ ∉ dom(ht) ∧ aₛ ∉ dom(ht)
∧ aₛ ∈ NonProcessAddress ∧ aᵣ ∈ NonProcessAddress

(P[a → a'.add(v₁)], ht, h[a → ha], ch, mq)
→ (P[a := skip], ht, h[a := ha[a' := ha(a′) ∪ {v₁}]], ch, mq)
if a′ ∈ dom(ha) ∧ ht(a′) = set

(P[a → a'.add(v₁)], ht, h[a → ha], ch, mq)
→ (P[a := skip], ht, h[a := ha[a' := ha(a′)@⟨v₁⟩]], ch, mq)
if a′ ∈ dom(ha) ∧ ht(a′) = 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 → send v to a2], ht, h[a → ha, a2 → ha2], ch, mq)
→ (P[a := skip], ht', h[a := ha'[a5 := ha(a5)@((v1, a2))]], a2 := ha2',
   ch[(a, a2) := ch((a, a2))@⟨v2⟩], mq)
if a2 ∈ ProcessAddress ∧ a5 = ha(a)(sent) ∧ isCopy(v, ha, ha, ht, v1, ha', ht')
  ∧ isCopy(v, ha', ha2, ht', v2, ha2', ht'')

// send to a set of processes
(P[a → send v to a'], ht, h[a → ha], ch, mq)
→ (P[a := for x in a': send v to x], ht, h[a := ha[a := ha(a)@((v, a'))]], ch, mq)
if ht(a') = set ∧ a5 = ha(a)(sent) ∧ (x is a fresh variable)

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

// message loss
(P, ht, h, ch[(a, a') → q], mq) → (P, ht, h, ch[(a, a') := q'], mq)
if (channel reliability is unreliable in the program configuration) ∧ (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') → q], mq) → (P, ht, h, ch[(a, a') := q'], mq)
if length(q) > 0 ∧ a5 = ha(a)(received) ∧ isCopy(first(q), ha, ha, ht, copy, ha', ht')
  ∧ receiveAtLabel(first(q), t, ht(a), ha') = S ∧ S' is a linearization of S

// wait without timeout clause
(P[a → ℓ await e1; s1 or · · · or en; sN], ht, h, ch, mq) → (P[a := s1], ht, h, ch, mq)
if length(mq(a)) = 0 ∧ i ∈ [1..N] ∧ h(a) : ht + e1 → true

// await with timeout clause, terminated by true condition
(P[a → ℓ await e1; s1 or · · · or en; sN timeout v; s], ht, h, ch, mq) → (P[a := s1], ht, h, ch, mq)
if length(mq(a)) = 0 ∧ i ∈ [1..N] ∧ h(a) : ht + e1 → true

// await with timeout clause, terminated by timeout (occurs non-deterministically)
(P[a → ℓ await e1; s1 or · · · or en; sN timeout v; s], ht, h, ch, mq) → (P[a := s1], ht, h, ch, mq)
if length(mq(a)) = 0 ∧ h(a) : ht + e1 → false ∧ · · · ∧ h(a) : ht + eN → false

// context rule for expressions
h(a) : ht + e → e'
(P[a → C[e]], ht, h, ch, mq) → (P[a := C[e']], ht, h, ch, mq)

// context rule for statements
(P[a → s], ht, h, ch, mq) → (P[a := s'], ht', h', ch', mq')
(P[a → C[s]], ht, h, ch, mq) → (P[a := C[s']], ht', h', ch', mq')

Fig. 14. Transition relation for statements, Part 2.
\[\text{Init} =\]
\[\{(P, ht, h, ch, mq) \in \text{State} \mid\]
\[\exists a_p \in \text{ProcessAddress},\]
\[a_r \in \text{NonProcessAddress},\]
\[a_s \in \text{NonProcessAddress}.\]
\[a_r \neq a_s\]
\[\land P = f_0[a_p := a_p \text{.main()}]\]
\[\land ht = f_0[a_p := \text{process}, a_r := \text{sequence}, a_s := \text{sequence}]\]
\[\land h = f_0[a_p := ha]\]
\[\land ch = (\lambda(a_1, a_2) \in \text{ProcessAddress} \times \text{ProcessAddress}. (\langle \rangle))\]
\[\land mq = (\lambda a \in \text{ProcessAddress}. (\langle \rangle))\]
\[\text{where } ha = f_0[a_p := o_p, a_r := (\langle \rangle), a_s := (\langle \rangle)]\]
\[o_p = f_0[\text{received} := a_r, \text{sent} := a_s]\]

Fig. 15. Initial states.

REFERENCES


