From Clarity to Efficiency for Distributed Algorithms *

Yanhong A. Liu  Scott D. Stoller  Bo Lin
Computer Science Department, Stony Brook University, Stony Brook, NY 11794, USA
{liu,stoller,bolin}@cs.stonybrook.edu

Abstract
This paper 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.

Keywords distributed algorithms, incrementalization, logic quantifications, operational semantics, program optimization, very high-level languages

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 pseudo-code 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.
• 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.

We have developed 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, and be executed as part of real applications, as in programming languages.

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

We describe 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 set queries. In general, however, translating nested quantifications simply into nested queries 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 non-trivial distributed algorithms. They can be incrementally expressed 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 very high-level language that combines the best of pseudocode, specification, and programming languages.
- 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 a very high-level language and the effectiveness of our optimizations.

### 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 [27] is so fundamental for distributed systems.

**Running example.** We use Lamport’s distributed mutual exclusion algorithm [27] 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 \( \Rightarrow \) in rule 5 (for comparing pairs of timestamps and process ids) and with the word “acknowledgment” added in rule 5 (for simplicity when omitting a commonly omitted [17, 39] 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 of entering and exiting a critical section while also responding to messages from other processes. It must also
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 $T_{m_i}:P_i \text{ requests resource}$ to every other process, and puts that message on its request queue, where $T_{m_i}$ is the timestamp of the message.

2. When process $P_j$ receives the message $T_{m_i}:P_i \text{ requests resource}$, 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 $T_{m_i}:P_i \text{ requests resource}$ 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 $T_{m_i}:P_i \text{ requests resource}$ message from its request queue.

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

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

Figure 1. Original description in English.

3. DistAlgo Language

To support distributed programming at a high level, four main concepts can be added to commonly used object-oriented programming languages, such as Java and Python: (1) processes as objects, and sending of messages, (2) yield points and waits for control flows, and handling of received messages, (3) synchronization conditions using high-level queries and 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 like threads 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 the following form. It defines a type of processes named $p$, by defining a class $p$ that extends class $\text{process}$. The $\text{process}_\text{body}$ is a set of method definitions and handler definitions, to be described.

```java
class p extends process:
    process_body
```

A special method $\text{setup}$ may be defined in $\text{process}_\text{body}$ for initially setting up data in the process before the execution starts. A special method $\text{run}()$ must be defined in $\text{process}_\text{body}$ for carrying out the main flow of execution, and a call $\text{start}()$ starts the process and executes the $\text{run}$ method. A special variable $\text{self}$ refers to the current process. A special field $\text{id}$ holds the id of the process.

A statement for process creation is of the following form. It creates $n$ new processes of type $p$, and assigns the single new process or set of new processes to variable $v$; expression $\text{node}_\text{exp}$ evaluates to a node (a host name or IP address plus a port number) or a set of nodes, specifying where the new processes will be created.

```java
v = n new p at node_exp
```

The number $n$ and the at clause are optional; the defaults are 1 and local node, respectively.

A statement for sending messages is of the following form. It sends the message that is value of expression $m_exp$ to the process or set of processes that is the value of expression $p_exp$.

```java
send m_exp to p_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 the following form, where identifier \( l \) is a label and is optional. It specifies that point in the program as a place where control may yield to handling of received messages.

\[
\text{-- \text{\textbf{l}:}}
\]

A handler definition, also called receive definition, is of the following form. It handles, at yield points labeled \( l_1, \ldots, l_j \), any un-handled message that matches some \( \text{mexp}_k \) sent from \( \text{pexp}_k \), where \( \text{mexp}_k \) and \( \text{pexp}_k \) are parts of a tuple pattern; previously unbound variables in a pattern are bound to the corresponding components in the value matched. The \text{handler\_body} is a sequence of statements to be executed for the matched messages.

\[
\text{receive \text{mexp}, from \text{pexp}, \ldots, \text{mexp}, from \text{pexp},}
\text{at \( l_1, \ldots, l_j \):}
\text{handler\_body}
\]

The \text{from} and \text{at} clauses are optional; the defaults are any process and all yield points, respectively. If the \text{from} clause is used, each message is automatically associated with the corresponding 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 of the following form. It waits for the value of Boolean-valued expression \( \text{bexp} \) to become true.

\[
\text{await \text{bexp}}
\]

A general, nondeterministic await statement is of the following form. It waits for any of the values of expressions \( \text{bexp}_1, \ldots, \text{bexp}_k \) to become true or a timeout after \( t \) seconds, and then nondeterministically selects one of statements \( \text{stmt}_1, \ldots, \text{stmt}_k \), \text{stmt} whose corresponding conditions are satisfied to execute. The \text{or} and \text{timeout} clauses are optional.

\[
\text{await \text{bexp}_1: \text{stmt}_1}
\text{or \ldots}
\text{or \text{bexp}_k: \text{stmt}_k}
\text{timeout \( t \): \text{stmt}}
\]

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

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 \text{receive} definitions and a \text{run} method containing only \text{await false}, or by writing just a \text{run} method containing only a while \text{true} loop whose body is a receive definition.

**Synchronization conditions using high-level queries.** Synchronization conditions and other conditions can be expressed using high-level queries—quantifications, comprehensions, and aggregates—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 theoretical efficiency of distributed algorithms is measured by 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 used aggregates in place of quantifications before we developed the method to systematically optimize quantifications. For example, we regularly expressed “\( \forall v \) is larger than all elements of \( \mathcal{S} \)” as \( \forall v \geq \max(\mathcal{S}) \) and either forgot to handle the case that \( \mathcal{S} \) is empty or handled it in ad hoc fashions. Naive use of aggregates 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 following two forms, 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 \textit{reachable} from a parameter—be a parameter or be the left-side variable of a membership clause whose right-side variables are reachable. Given values of parameters, the query returns \textit{true} iff for some or all, respectively, values of the variables that satisfy all membership clauses, \( \text{bexp} \) evaluates to \textit{true}. When an existential quantification returns \textit{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 \( \text{bexp} \).

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

To indicate any variable \( x \) on the left side of a membership clause to be a parameter, add prefix \( = x \) to \( x \). Notation \( =x \) means a value that is equal to the value of parameter
x; it is equivalent to using a fresh variable y instead and adding a conjunct y=x in condition bexp.

- A comprehension is a query of the following form plus a set of parameters. Given values of parameters, the query returns the set of values of exp for all values of variables that satisfy all membership clauses vi in exp, and condition bexp.

\[ \{ \text{exp: } v_1 \text{ in } sexp_1, \ldots, v_k \text{ in } sexp_k \mid bexp \} \]

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

• Aggregates are of the form \text{agg}(ssexp), where agg is an operation, such as \text{size}, \text{sum}, or \text{max}, specifying the kind of aggregation over the set value of sexp.

• In the query forms above, each \text{v}_i can also be a tuple pattern \text{t}_i. Previously unbound variables in \text{t}_i are bound to the corresponding components in the matched elements of the value of sexp_i. We omit \mid bexp when bexp is true.

We use \{\} for empty set; use \text{a.add}(x) and \text{a.del}(x) for element addition and deletion, respectively; and use \text{x in a} and \text{x not in a} for membership test and its negation, respectively. We assume that hashing is used in implementing sets, and the expected time of set membership tests and updates involving one element is \(O(1)\).

DistAlgo has two built-in sequences, \text{received} and \text{sent}, containing all messages received and sent, respectively, by a process.

- Sequence \text{received} is updated only at yield points. An arrived message \text{m} for which the program contains a matching receive definition is added to \text{received} when the program reaches a yield point where \text{m} is handled, and all matching message handlers associated with that yield point are executed for \text{m}. An arrived message for which the program contains no matching receive definitions is added to \text{received} at the next yield point. The sequence \text{sent} is updated at each \text{send} statement.

- We use \text{received(m from p)} as a shorthand for \text{m from p in received}; from p is optional, but when specified, each message in received is automatically associated with the corresponding sender. We use \text{sent(m to p)} as a shorthand for \text{m to p in sent}; to p is optional, but when specified, p is the process to which \text{m} was sent as specified in the corresponding \text{send} statement.

If implemented straightforwardly, \text{received} and \text{sent} can create a huge memory leak, because they can grow unbounded, preventing their use in practical programming.

**Configuration.** One can specify channel types, handling of messages, setup for starting processes, 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.

The following statement configures all channels to be \text{fifo}. Other options for channel include \text{reliable} and \text{reliable, fifo}. When these options are specified, TCP is used process communication; otherwise, UDP is used.

\text{configure channel = fifo}

In general, separate channel types can be specified for communication among any set of processes; the default is for communication among all processes.

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

\text{configure handling = all}

means that all matching received messages that are not yet handled must be handled before execution of the main flow of control continues past any 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 [15, 27, 40] are used in many distributed algorithms. One can specify the logical clock, e.g., Lamport clock, that is used:

\text{configure clock = Lamport}

It configures sending and receiving of messages to update the clock appropriately. One can call \text{logical_clock()} to get the value of the clock.

**Other language constructs.** For other constructs, we use those in high-level object-oriented languages. We mostly use Python syntax (indentation for scoping, ‘:’ for separation, ‘#’ for comments, etc.), for succinctness, except with a few conventions from Java (keyword \text{extends} for subclass, keyword \text{new} for object creation, and omission of \text{self}, equivalent of \text{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 \text{cs} and the two \text{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 \text{main}, which configures the system to run (lines 25-30). Method \text{cs} and the two \text{receive} definitions are executed when needed and follow the five rules in Figure 1 (lines 5-21).

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 contrast 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,
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 cs(task): # for calling task() in CS
6       self.c = logical_clock()  # 1 in Fig 1
7       send ('request', c, id) to s #
8       q.add(('request', c, id)) #
9           # wait for own req < others in q #
10      # and for acks from all in s #
11      await each ('request', c2, p2) in q | # 5 in Fig 1
12      (c2,p2) != (c,id) implies (c,id) < (c2,p2) #
13      and each p2 in s | #
14          some received('ack',c2,p2) | c2 > c #
15      task() # critical section
16      q.del(('request', c, id)) # 3 in Fig 1
17      send ('release', logical_clock(), id) to p #
18      # use reliable and FIFO channel
19      if some ('request', c2, p2) in q: #
20          q.del(('request', c2, p2)) #
21      receive ('release', _, p2): # 4 in Fig 1
22      for p in ps: p.start() # start each proc, call method run
23      for p in ps: p.setup(ps-{p}) # pass to each proc other procs
24      ps = 50 new P # create 50 processes of P class
25      configure clock = Lamport # use Lamport clock
26      configure channel = (reliable, fifo) # use reliable and FIFO channel
27      ps = 50 new P # create 50 processes of P class
28      for p in ps: p.setup(ps-{p}) # pass to each proc other procs
29      for p in ps: p.start() # start each proc, method run
30      # other tasks of the application

31 def run(): # main method for the process
32      ... # may do non-CS tasks of the proc
33 def task(): ... # define critical section task
34      cs(task) # call cs to do task in CS
35      ... # other tasks of the application
36 def main(): # main method for the application
37      ... # other tasks of the application
38      configure channel = (reliable, fifo) # use reliable and FIFO channel
39      configure clock = Lamport # use Lamport clock
40      ps = 50 new P # create 50 processes of P class
41      for p in ps: p.start() # start each proc, call method run
42      ... # other tasks of the application

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

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. High-level programming constructs, such as new p, can easily be compiled into loops.

Sending a message m to a process or set of processes, 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) to be executed. 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 1 is compiled into a call to a message handler method 1() that updates the sequence received, if it is used in the program, and executes the bodies of the receive definitions whose at clause includes 1. Precisely:

1. 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 pattern appropriately, and executes the statement in the body of this receive definition.

2. Method 1() compiled for yield point 1 does the following: for each message m from p in the queue of messages not yet handled, (1) if m matches a message pattern in a receive definition whose at clause includes 1, then execute received.add(m from p) if received is used in the program and call the methods generated from the receive definitions whose at clause includes 1; (2) if m does not match any message pattern in any receive definition, then execute received.add(m from p) if received is used in the program. In both these cases, remove m from the message queue afterward.

An await statement can be compiled into a synchronization using busy-waiting or blocking. For example, for busy-waiting, a statement await bexp that immediately follows a label l is compiled into a call l() followed by while not bexp: l().

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

5. Incrementalizing expensive synchronizations

Incrementalization transforms expensive computations into efficient incremental computations with respect to updates to the values on which the computations depend. It (1) identifies all expensive queries, (2) determines all updates to the parameters of these queries, and (3) transforms the queries and updates into efficient incremental computations. Much of this has been studied previously.

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 drastic reduction of both time and space complexities.

Expensive computations using quantifications. Expensive computations in general involve repetition, including loops, recursive functions, comprehensions, aggregates, and quantifications over collections. Loops were studied most; less for recursive functions and comprehensions, and least for quantifications, basically corresponding to how frequently each construct has 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 incurs 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 aggregate queries that can be optimized systematically using previously studied methods. However, a quantification can be converted into multiple forms of aggregate queries. 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 queries 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 aggregate query if a witness is not needed; if a witness is needed, we incrementally compute the set of witnesses.

Converting quantifications to aggregate queries. We present all converted forms here and describe which forms to use after we discuss the updates that must be handled. The process to develop them was nontrivial, even though the end results look simple. The correctness of all rules presented are proved 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 queries that use size aggregates. These rules are general because expr can be any Boolean expression, but they are for converting single quantifications. Nested quantifications could be converted one at a time from inside out, but the results can be much more complicated than necessary. For example,

\[ \text{each } x \text{ in } s \mid \text{some } y \text{ in } t \mid \text{expr} \]

would be converted using rule 1 to

\[ \text{each } x \text{ in } s \mid \text{size(} \{ \text{y in } t \mid \text{expr} \} \text{) == 0} \]

and then using rule 2 to

\[ \text{size(} \{ x \text{ in } s \mid \text{size(} \{ y \text{ in } t \mid \text{expr} \} \text{) == 0} \} \text{) == size(s)} \]

Table 2 shows general rules for converting nested quantifications into equivalent, but non-nested, queries that use size aggregates. 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>
<thead>
<tr>
<th>Quantification</th>
<th>Using Aggregate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 some x in s \mid expr</td>
<td>size({ x in s \mid expr } != 0)</td>
</tr>
<tr>
<td>2 each x in s \mid expr</td>
<td>size({ x in s \mid expr } == size(s))</td>
</tr>
<tr>
<td>3</td>
<td>size({ x in s \mid \neg expr } == 0)</td>
</tr>
</tbody>
</table>

Table 1. Rules for converting single quantifications.

Table 3 shows general rules for converting single quantifications with a single order comparison into equivalent queries that use max and min aggregates. These rules are useful because single quantified order comparison, when there are no element deletions, can be computed more efficiently, with a constant instead of linear space overhead.

<table>
<thead>
<tr>
<th>Existential</th>
<th>Using Aggregate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 some x in s \mid y &lt;= x</td>
<td>s != {} and y &lt;= max(s)</td>
</tr>
<tr>
<td>2 each x in s \mid y &gt;= y</td>
<td>s != {} and y &gt;= min(s)</td>
</tr>
<tr>
<td>3 each x in s \mid y &lt; x</td>
<td>s != {} and y &lt; max(s)</td>
</tr>
<tr>
<td>4 each x in s \mid y &gt; x</td>
<td>s != {} and y &gt; min(s)</td>
</tr>
<tr>
<td>5 each x in s \mid x &lt;= y</td>
<td>s != {} and y &gt;= min(s)</td>
</tr>
<tr>
<td>6 each x in s \mid x &gt;= y</td>
<td>s != {} and y &lt;= max(s)</td>
</tr>
<tr>
<td>7 each x in s \mid y &lt; x</td>
<td>s != {} and y &lt; max(s)</td>
</tr>
<tr>
<td>8 each x in s \mid y &gt; x</td>
<td>s != {} and y &gt; min(s)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Universal</th>
<th>Using Aggregate</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 each x in s \mid y &lt;= x</td>
<td>s != {} and y &lt;= max(s)</td>
</tr>
<tr>
<td>10 each x in s \mid x &gt;= y</td>
<td>s != {} and y &gt;= min(s)</td>
</tr>
<tr>
<td>11 each x in s \mid y &gt;= y</td>
<td>s != {} and y &gt;= max(s)</td>
</tr>
<tr>
<td>12 each x in s \mid y &lt; x</td>
<td>s != {} and y &lt; min(s)</td>
</tr>
<tr>
<td>13 each x in s \mid x &gt; y</td>
<td>s != {} and x &gt;= min(s)</td>
</tr>
<tr>
<td>14 each x in s \mid x &lt; y</td>
<td>s != {} and x &lt; max(s)</td>
</tr>
<tr>
<td>15 each x in s \mid y &gt; x</td>
<td>s != {} and y &gt; min(s)</td>
</tr>
<tr>
<td>16 each x in s \mid x &lt; y</td>
<td>s != {} and x &gt; max(s)</td>
</tr>
</tbody>
</table>

Table 3. Rules for single quantified order comparison.

Boolean combinations of order comparisons and other conditions can be transformed first into quantifications each involving at most one order comparison at a time. Table 4 shows general rules for decomposing combinations of conditions in general quantifications, to extract quantifications each involving a single order comparison. For example,

\[ \text{each } x \text{ in } s \mid \text{expr} \implies y < x \]

can be converted using rule 6 to

\[ \text{each } x \text{ in } \{ x \text{ in } s \mid \text{expr} \} \mid y < x \]

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

\[ \{ x \text{ in } s \mid \text{expr} \} == \{ y \text{ in } \{ x \text{ in } s \mid \text{expr} \} \mid y < x \} \]
Using Aggregate

some \( x \) in \( s \) | each \( y \) in \( t \) | bexp
size({(x,y): x in \( s \), y in \( t \) | bexp}) != 0

Decomposed Quantifications

size({(x,y): x in \( s \), y in \( t \) | bexp}) == size({(x,y): x in \( s \), y in \( t \)})

size({(x,y): x in \( s \), y in \( t \) | not bexp}) != size({(x,y): x in \( s \)})

each \( x \) in \( s \)

each \( x \) in \( s \) | each \( y \) in \( t \) | bexp

each \( x \) in \( s \) | some \( y \) in \( t \) | bexp

some \( x \) in \( s \) | some \( y \) in \( t \) | bexp

size({(x,y): x in \( s \), y in \( t \) | bexp}) == size({(x,y): x in \( s \), y in \( t \)}) == size({(x,y): x in \( s \), y in \( t \) | not bexp}) == 0

Table 2. Rules for converting nested quantifications.

<table>
<thead>
<tr>
<th>Quantification</th>
<th>Decomposed Quantifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>some ( x ) in ( s )</td>
<td>some ( x ) in {x in ( s )</td>
</tr>
<tr>
<td>some ( x ) in ( s )</td>
<td>some ( x ) in ( s )</td>
</tr>
<tr>
<td>some ( x ) in ( s )</td>
<td>some ( x ) in ( s )</td>
</tr>
<tr>
<td>each ( x ) in ( s )</td>
<td>each ( x ) in ( s )</td>
</tr>
<tr>
<td>each ( x ) in ( s )</td>
<td>each ( x ) in {x in ( s )</td>
</tr>
<tr>
<td>each ( x ) in ( s )</td>
<td>each ( x ) in {x in ( s )</td>
</tr>
<tr>
<td>each ( x ) in ( s )</td>
<td>each ( x ) in {x in ( s )</td>
</tr>
<tr>
<td>each ( x ) in ( s )</td>
<td>each ( x ) in {x in ( s )</td>
</tr>
</tbody>
</table>

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

Updates caused by message passing. Parameters of a query are variables in the query whose values may affect the query result. Updates to a parameter are operations that may change the value of the parameter. The most common updates are assignments, \( v = \text{exp} \), which is an update to \( v \). Other updates can all be expressed as assignments. For objects, all updates can be expressed as field assignments, \( \text{obj}.f = \text{exp} \). For collections, all updates can be expressed as initialization to empty and element additions and removals.

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., [19, 34].

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 [34–36, 44].

For aggregate queries converted from quantifications, algebraic properties of the aggregate operations 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 aggregate query once per update. Additionally, if \( \max \) and \( \min \) aggregates are used and there are no element deletions from the sets queried, the space overhead is constant. Note that if \( \max \) and \( \min \) are used naively and there are element deletions, there would be an overhead of \( O(n) \) space and \( O(\log n) \) update time from using more sophisticated data structures to maintain the \( \max \) or \( \min \) under element deletion [12, 20, 57, 58].

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

- For nested quantifications, the rules in Table 2 are used.
- 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 no deletions from 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 constant.
- Multiple ways of conversion may be possible: for universal quantifications using rules 2 and 3 in Table 1 and rules 4 and 5 in Table 2, for nested quantifications with two or more alternations using rules 2 and 3 in Table 2 (each way of conversion corresponds to a choice of which two alternating quantifiers to eliminate using one of the rules), and for quantifications with symmetric ways of decomposing combinations of conditions using rules 1, 5, and 6 in Table 4. Our method transforms in all these ways, obtains the time and space complexities for each result, and chooses one with the best time and then space.

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 given in the table and insert at each update the corresponding maintenance; a fresh variable is introduced for each different query.

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

Table 5. Incrementalization rules for size and for max. The rule for min is similar to the rule for max.

The overall incrementalization algorithm [34, 35, 44] (1) introduces new variables to store the results of expensive queries and subqueries, as well as appropriate additional values, (2) transforms the queries and subqueries to use the stored query results and additional values, and (3) transforms updates to query parameters to also do incremental maintenance of the stored query results and additional values.

If queries are nested, inner queries are transformed before outer queries. Note that a comprehension such as \( \{ x \in s \mid \text{bexp} \} \) is incrementalized with respect to changes to parameters of Boolean expression \( \text{bexp} \) as well as addition and removal of elements of \( s \); if \( \text{bexp} \) 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 \( (\text{ack'}, c2, p2) \) in received is used in place of \( \text{received}(\text{ack'}, c2, p2) \).

```
each (\text{request'}, c2, p2) \in q \mid 
(c2, p2) != (c, id) \text{ implies } (c, id) < (c2, p2) 
and each p2 in s \mid 
some (\text{ack'}, c2, =p2) \in \text{received} \mid c2 > c
```

Converting quantifications into aggregates 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:

\[
\text{size}(\{(\text{request'}, c2, p2) \in q \mid (c, id) > (c2, p2)\}) = 0
\]

and
\[
\text{size}(\{p2: p2 \in s, (\text{ack'}, c2, =p2) \in \text{received} \mid c2 > c\}) = \text{size}(s)
\]

Updates to parameters of the first conjunct are additions and removals of requests to and from \( q \), and also assignment to \( c \). Updates to parameters of the second conjunct are additions of \( \text{ack} \) messages to \( \text{received} \), and assignment to \( c \), after the initial assignment to \( s \).

Incremental computation [34–36, 44] introduces variables to store the values of all three aggregates in the converted query, transforms the aggregates to use the introduced variables, and incrementally maintains the stored values at each of the updates, yielding the following:

- For the first conjunct, store the set value and the size value in two variables, say \( \text{earlier} \) and \( \text{count} \), respectively; when \( c \) is assigned a new value, let \( \text{earlier} \) be \( q \) and let \( \text{count1} \) 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 \( c \) is defined and \( (c, id) > (c2, p2) \) holds, add the request to \( \text{earlier} \) and increment \( \text{count1} \) by 1, taking \( O(1) \) time; similarly for deletion from \( q \).

Note that when \( (\text{request'}, c, id) \) in particular is added to or removed from \( q \), \( \text{earlier} \) and \( \text{count1} \) are not updated, because \( (c, id) > (c, id) \) is trivially false.

- For the second conjunct, store the set value and the two size values in three variables, say \( \text{responded} \), \( \text{count2} \), and \( \text{total} \), respectively; when \( s \) is initialized in setup, assign \( \text{total} \) the size of \( s \), taking \( O(|s|) \) time, done only once for each process; when \( c \) is assigned a new value, let \( \text{responded} \) be \( \{\} \), and let \( \text{count2} \) be 0, taking \( O(1) \) time; when an \( \text{ack} \) message is added to \( \text{received} \), if the associated conditions hold, increment \( \text{count2} \) by 1, taking \( O(1) \) time.

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 \( \text{Lamport\_clock()} \) returns a timestamp larger than all timestamps of messages previously received, and thus at the assignment to \( c \), we have that earlier is \( q \) and \( \text{responded} \) is \( \{\} \).
Figure 3 shows the optimized program after incrementalization of the synchronization condition on lines 10-11 in Figure 2. All commented lines are new except that the synchronization condition in the `await` statement is simplified. The synchronization condition now takes \(O(1)\) time, compared with \(O(|s|^2)\) if implemented straightforwardly. The trade-off is the amortized \(O(1)\) time overhead at updates to \(c\) and \(q\) and on receiving of ack messages.

Note that the sequence `received` used in the synchronization condition in Figure 2 is no longer used after incrementalization. All values needed for evaluating the synchronization condition are stored in new variables introduced: `earlier`, `count1`, `responded`, `count2`, and `total`, a drastic space improvement from unbounded for `received` to linear in the number of processes.

```python
1 class P extends Process:
2     def setup(s):
3         self.s = s
4         self.q = {}
5         self.total = size(s) # total num of other procs
6     def cs(task):
7         -- request
8         self.c = logical_clock()
9         self.earlier = q # set of pending earlier reqs
10     def setup(s):
11         self.count1 = size(earlier) # num of pending earlier reqs
12         self.responded = {} # set of responded procs
13         self.count2 = 0 # num of responded procs
14         self.responded = {} # set of responded procs
15         self.earlier = q # set of pending earlier reqs
16         self.count1 = self.responded = {} # init counters
17         await count1 == 0
18         and count2 == total # use maintained results
19     task()
20     -- release
21     q.del(('request', c, id))
22     send ('release', logical_clock(), id) to s
23     receive ('request', c2, p2):
24         if (c,id) < min({(c2,p2): ('request',c2,p2) in q
25             and (c2,p2) != (c,id)}) == {} # first conjunct
26             q.add(('request', c2, p2))
27             send ('ack', logical_clock(), id) to p2
28         if c != undefined: # if c is defined
29             if c > c2: # comparison in conjunct 1
30                 q.del(('request', c2, p2))
31                 send ('ack', logical_clock(), id) to p2
32             if p2 not in responded: # if not responded already
33             responded.add(p2) # add to responded
34             count2 += 1 # increment count2
35             if some ('request', c2, p2) in q:
36                 if c != undefined: # if c is defined
37                     if (c,id) > (c2,p2): # comparison in conjunct 1
38                         q.del(('request', c2, p2))
39                     task() # membership in conjunct 2
40                 if (c,id) in earlier: # if in earlier
41                 earlier.del(('request', c2, p2)) # delete it
42                 count1 -= 1 # decrement count1
43                 q.del(('request', c2, p2))
44     Figure 3. Optimized program after incrementalization. Definitions of `run` and `main` are as in Figure 2.

Example with naive use of aggregate \(\min\). Note that, for the example above, 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 can in fact be implemented with a bit for each process using previous work on data structure selection method [31, 43].

We show that if \(\min\) is used naively, a more sophisticated data structure [12, 20, 57, 58] supporting priority queue is in fact needed, incurring an \(O(\log n)\) time update instead of \(O(1)\) time in the example above. Additionally, for a query using \(\min\) to be correct, special care must be taken to deal with the case when the argument to \(\min\) is empty because \(\min\) is not defined on an empty argument.

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

\[
\begin{align*}
(c, id) &< \min((c,2,p2): ('request',c,2,p2) in q \\
&\quad | (c,2,p2) != (c, id) \} == {} \\
\end{align*}
\]

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

\[
\begin{align*}
(c, id) &< \min((c,2,p2): ('request',c,2,p2) in q \\
&\quad | (c,2,p2) != (c, id) \} == {} \\
\end{align*}
\]

However, that is incorrect, because the argument of \(\min\) may be empty, in which case \(\min\) is not defined.

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

\[
\begin{align*}
\{ (c,2,p2): ('request',c,2,p2) in q &\quad | (c,2,p2) != (c, id) \} == {} \\
\text{or} &
\end{align*}
\]

In fact, the original universal quantification in the first conjunct in the `await` statement can be converted exactly to this disjunction by using rule 6 in Table 4 and then rule 13 in Table 3. Our method does not consider this conversion at all because it leads to a worse resulting program.

Figure 4 shows the resulting program after incrementalization of the synchronization condition that uses the disjunction above, where \(ds\) stores the argument set of \(\min\). 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 long complex code for maintaining the data structure \(ds\) is not included; it is in fact similar to that program except that \(ds\) is used and maintained instead of `earlier` and its count, and that \(q\) is removed because it is captured in \(ds\). This program is still a drastic improvement over the original program in Figure 2, with the synchronization condition in \(O(1)\) time and with `receive` removed, except that maintaining \(ds\) for incrementalizing \(\min\) under element addition to and deletion from \(q\) takes \(O(\log |s|)\) time, as opposed to \(O(1)\) time for maintaining `earlier` and its count in Figure 3.

Simplifications to the original algorithm. Consider the original algorithm in Figure 2. Note that incrementalization
1 class P extends Process:
2   def setup(s):
3       self.s = s
4   self.total = size(s)  # total num of other procs
5   self.ds = new DS()  # data structure for main
6       # requests by other procs
7   def cs(task):
8       -- request
9       self.c = logical_clock()
10      self.responded = {}  # set of responded procs
11      self.count = 0  # num of responded procs
12      send ('request', c, id) to s
13      await (ds.is_empty() or (c,id) < ds.min())
14      and count == total  # use maintained results
15      task()
16      -- release
17      send ('release', Lamport_clock(), id) to s
18      ds.add((c2,p2))  # add to data structure
19      send ('ack', logical_clock(), id) to p2
20      receive ('ack', c2, p2):  # new message handler
21      if c2 > c:  # comparison in conjunct 2
22      if p2 in s:  # membership in conjunct 2
23      if p2 not in responded:  # if not responded already
24      responded.add(p2)  # add to responded
25      count += 1  # increment count
26      receive ('release', _, p2):
27      if some ('request', c2, p2) in q:
28      ds.del((c2,p2))  # remove from data structure

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

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 that updates to q for a process’s own request do not affect the only use of q, on line 10, in Figure 3, so we can remove them in Figure 3 and remove them on lines 9 and 14 in Figure 2 as well as the test (c2,p2) != (c,id), which becomes always true, in the synchronization condition.

Furthermore, note that the remaining updates to q in Figure 2 are merely maintaining pending requests by others, so we can remove lines 4, 17, 20, 21, and the entire receive definition for release messages, by using c in the release message on line 15 and using, for the first conjunct in the await statement,

each received('request',c2,p2) | not received('release',c2,p2)
implies (c,id) < (c2,p2)

Figure 5 shows the resulting simplified algorithm. Incrementalizing this program yields essentially the same programs as in Figures 3 and 4, except that (1) the send statement for release messages uses c instead of logical_clock(), (2) the receive clause for release messages uses variable c2 instead of _, and (3) condition if some (‘request’, c2, =p) in q is removed.

The exact three changes above can also be applied to the original algorithm in Figure 2 and even Figure 1. This is an improvement, because in the original algorithm, the logical_clock() value sent in release messages is not used in the handler for received release messages, and the handler has to locate the matching request messages.

Indeed, rule 4 in Figure 1 has an ambiguity, because “any Tm, P, requests resource message” can be interpreted as either “any one” or “all”. If one understands that each process can make only one request at a time, then either interpretation is correct. However, this is not clear in Figure 1 because the five rules are not organized, and each rule could in general be executed repeatedly before another rule is executed, which is in fact necessary for rules 2 and 4. If rule 1 could be executed repeatedly, i.e., a process can make multiple requests at a time, then either interpretation for rule 4 would be incorrect, because only the matching request should be removed.

1 class P extends Process:
2   def setup(s):
3       self.s = s
4   def cs(task):
5       -- request
6       self.c = logical_clock()
7       send ('request', c, id) to s
8       await each received('request',c2,p2) |
9       not received('release',c2,p2)
10      implies (c,id) < (c2,p2)
11      and each p2 in s |
12      some received('ack','c2,'p2') |
13      c2 > c
14      task()
15      -- release
16      send ('release', c, id) to s
17      receive ('request', _, p2):  # remove from data structure
18      send ('ack', logical_clock(), id) to p2

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

6. Implementation and experiments

We developed a prototype implementation of the compiler and optimizations for DistAlgo, as described previously [38]. A more extensive implementation of the compiler has since been developed [30]. It takes DistAlgo programs written in extended Python and generates executable Python code. It optionally interfaces with an incrementalizer to apply incrementalization before generating code. A more extensive implementation of incrementalization is also being developed. Experiments with these new implementations confirm the analyzed time and space complexity improvements, as in previous experiments [38].

We have used DistAlgo to implement a variety of well-known distributed algorithms, including different algorithms for distributed mutual exclusion, leader election, and atomic commit, as well as Paxos, Byzantine Paxos, and multi-Paxos, as described previously [38]. This has allowed us to discover several improvements to correctness and efficiency aspects of some of the algorithms [37]. Additionally, students in distributed algorithms courses have used DistAlgo in dozens of course projects, implementing the core of network protocols and distributed graph algorithms; distributed coordination services Chubby and Zookeeper; distributed
hash tables Kademlia, Tapestry, Chord, and Dynamo; distributed file systems HDFS, GFS, and BFS; distributed databases Bigtable, Cassandra, and Megastore; distributed processing platform MapReduce; and others. These experiences help confirm that DistAlgo allows complex distributed algorithms and services to be implemented much more easily than commonly used languages such as C++ and Java.

7. Related work

A wide spectrum of languages and notations have been used to describe distributed algorithms, e.g., [5, 17, 26, 28, 29, 39, 49–51, 56]. At one end, pseudocode with English is used, e.g., [26], which well 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 [24, 39], 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 [49–51] and Lamport’s PlusCal [29]. However, all of these languages and notations still lack concepts and mechanisms for building real distributed applications, and most of the languages are not executable at all.

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 [41]. 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. Some programming languages, such as Erlang [14], based on the actor model [2], have support for message passing and process management built into the language. They all lack constructs for expressing control flows and complex synchronization conditions at a much higher level, because these high-level constructs are extremely difficult to implement efficiently. DistAlgo’s construct for declaratively and precisely specifying yield points for message handlers is a new feature that we have not seen in other languages.

There has been much work on generating executable implementations from formal specifications, e.g., from process algebras [23], I/O automata [18], Unity [21], and Seuss [25], as well as from more recently proposed high-level languages for distributed algorithms, e.g., Datalog-based languages Meld [4], Overlog [3], and Bloom [9], and a logic-based language EventML [10, 46]. An operational semantics was also studied recently for a variant of Meld, called Linear Meld, that allows updates to be encoded more conveniently than Meld by using linear logic [13]. 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 widely used languages.

Incrementalization has been studied extensively, e.g., [31, 48], both done systematically based on languages, and routinely applied in ad hoc fashions to specific problems. However, all systematic incrementalization methods based on languages have been for centralized sequential programs, e.g., for set languages [22, 35, 44], recursive functions [1, 32, 47], logic rules [33, 53], and object-oriented languages [34, 42, 52]. This work is the first to extend incrementalization to distributed programs, where all sending and receiving of messages are systematically transformed into updates to message history sequences. 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 databases, despite extensive work on efficient implementation of high-level queries, efficient implementation of universal quantification has only been studied in limited scope or for extremely restricted query forms, e.g., [6–8, 11]. In logic programming, implementations of universal quantification are all based on variants of brute-force Lloyd-Topor transformations, e.g., [16, 45]; even state-of-the-art logic programming systems, e.g., [55] 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 aggregate queries that can be most efficiently incrementalized using the best previous methods.

To conclude, this paper 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.

Acknowledgments

We are grateful to the following people for their helpful comments and discussions: Ken Birman, Andrew Black, Jon Brandvein, Wei Chen, Ernie Cohen, John Field, Georges Gonthier, Leslie Lamport, Nancy Lynch, Lambert Meertens, Stephan Merz, Don Porter, Michel Raynal, John Reppy, Gun Sirer, Doug Smith, Robbert van Renesse, and anonymous reviewers.
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 [54, 59].

A.1 Abstract Syntax

The abstract syntax is defined in Figures 6 and 7. We use some syntactic sugar in sample code. Specifically, (1) we use infix notation for some binary operators, such as and is; (2) in comprehensions, if the Boolean expression is true, we elide “| true”; (3) in await statements, if the statement in one of the clauses is skip, we elide “: skip”.

Notation used in the grammar.

- 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.
- | separates alternatives.
- * after a non-terminal means “0 or more occurrences”
- + after a non-terminal means “1 or more occurrences”

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 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. The semantics of await statements not preceded by explicitly specified labels is given by translation. Specifically, for each await statement whose associated label is the empty string, we generate a fresh label name ℓ, associate ℓ with that await statement (by replacing the empty string with ℓ in that await statement), and insert ℓ in every at-clause in the class containing the await statement.
4. The boolean operators and and is; (2) in comprehensions, if the Boolean expression is true, we elide “| true”; (3) in await statements, if the statement in one of the clauses is skip, we elide “: skip”.

Notation used in the grammar.

- 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.
- | separates alternatives.
- * after a non-terminal means “0 or more occurrences”
- + after a non-terminal means “1 or more occurrences”

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 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. The semantics of await statements not preceded by explicitly specified labels is given by translation. Specifically, for each await statement whose associated label is the empty string, we generate a fresh label name ℓ, associate ℓ with that await statement (by replacing the empty string with ℓ in that await statement), and insert ℓ in every at-clause in the class containing the await statement.
4. The boolean operators and and is; (2) in comprehensions, if the Boolean expression is true, we elide “| true”; (3) in await statements, if the statement in one of the clauses is skip, we elide “: skip”.

Notation used in the grammar.

- 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.
- | separates alternatives.
- * after a non-terminal means “0 or more occurrences”
- + after a non-terminal means “1 or more occurrences”

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 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. The semantics of await statements not preceded by explicitly specified labels is given by translation. Specifically, for each await statement whose associated label is the empty string, we generate a fresh label name ℓ, associate ℓ with that await statement (by replacing the empty string with ℓ in that await statement), and insert ℓ in every at-clause in the class containing the await statement.
4. The boolean operators and and is; (2) in comprehensions, if the Boolean expression is true, we elide “| true”; (3) in await statements, if the statement in one of the clauses is skip, we elide “: skip”.

Notation used in the grammar.

- 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.
- | separates alternatives.
- * after a non-terminal means “0 or more occurrences”
- + after a non-terminal means “1 or more occurrences”

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 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. The semantics of await statements not preceded by explicitly specified labels is given by translation. Specifically, for each await statement whose associated label is the empty string, we generate a fresh label name ℓ, associate ℓ with that await statement (by replacing the empty string with ℓ in that await statement), and insert ℓ in every at-clause in the class containing the await statement.
4. The boolean operators and and is; (2) in comprehensions, if the Boolean expression is true, we elide “| true”; (3) in await statements, if the statement in one of the clauses is skip, we elide “: skip”.

Notation used in the grammar.

- 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.
- | separates alternatives.
- * after a non-terminal means “0 or more occurrences”
- + after a non-terminal means “1 or more occurrences”

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 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.
Program ::= Configuration ProcessClass* Method

ProcessClass ::= class ClassName extends ClassName: Method* ReceiveDef*

ReceiveDef ::=  
| receive ReceivePattern+ at LabelName+ : Statement  
| receive ReceivePattern+ : Statement

ReceivePattern ::= Pattern from InstanceVariable

Method ::=  
def MethodName(Parameter*) Statement  
defun MethodName(Parameter*) Expr

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

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

Tuple ::= (Expr*)

---

**Figure 6.** Abstract Syntax, Part 1

7. Comprehensions are statically eliminated as follows. The comprehension \( \ell x = \{ e \mid x_1 \text{ in } e_1, \ldots, x_n \text{ in } e_n, \mid b \} \), where \( \ell \) is a label and each \( x_i \) is a pattern, is replaced with

\[
\ell \ x = \text{new } \text{Set}() \\
\text{for } x_1 \text{ in } e_1:  
\text{...} \\
\text{for } x_n \text{ in } e_n:  
\text{if } b:  
\text{x.add(e)}
\]

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

9. Iterators containing tuple patterns are rewritten as iterators without tuple patterns, as follows.
UnaryOp ::= | not // boolean negation
| isTuple // test whether a value is a tuple
| len // length of a tuple

BinaryOp ::= is // identity-based equality, as in Python
| plus // sum
| select // select(t,i) returns the i’th component of tuple t

Pattern ::= InstanceVariable | TuplePattern

TuplePattern ::= (PatternElement*)

PatternElement ::= Constant | InstanceVariable | =InstanceVariable

Iterator ::= Pattern in Expr

AnotherAwaitClause ::= or Expr : Stmt

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

ClassName ::= ...
MethodName ::= ...
Parameter ::= self | ...
InstanceVariable ::= Expression.Field
Field ::= ...
Label ::= emptyString | LabelName
LabelName ::= ...
Literal ::= BooleanLiteral | IntegerLiteral | ...
BooleanLiteral ::= true | false
IntegerLiteral ::= ...

Figure 7. Abstract Syntax, Part 2

- Consider the universal quantification each \((e_1, \ldots, e_n)\) in \(s \mid b\). Let \(\theta\) be the substitution that replaces \(e_i\) with \(\text{select}(x,i)\) for each \(i\) such that \(e_i\) is a variable not prefixed with “\(=\)”. Let \(\{j_1, \ldots, j_m\}\) contain the indices of the constants and the variables prefixed with “\(=\)” in \((e_1, \ldots, e_n)\). Let \(x\) be a fresh variable. The quantification is rewritten as some \(x\) in \(s \mid \text{isTuple}(x)\) and \(\text{len}(x)\) is \(n\) and \(\text{select}(x,j_1)\) is \(e_{j_1}\) and \(\ldots\) and \(\text{select}(x,j_m)\) is \(e_{j_m}\) and \(b_\theta\).

- Consider the existential quantification some \((e_1, \ldots, e_n)\) in \(s \mid b\). Let \(\theta\) be the substitution that replaces \(e_i\) with \(\text{select}(x,i)\) for each \(i\) such that \(e_i\) is a variable not prefixed with “\(=\)”. Let \(\{j_1, \ldots, j_m\}\) contain the indices of the constants and the variables prefixed with “\(=\)” in \((e_1, \ldots, e_n)\). Let \(x\) be a fresh variable. The quantification is rewritten as some \(x\) in \(s \mid \text{isTuple}(x)\) and \(\text{len}(x)\) is \(n\) and \(\text{select}(x,j_1)\) is \(e_{j_1}\) and \(\ldots\) and \(\text{select}(x,j_m)\) is \(e_{j_m}\) and \(b_\theta\).

- Consider the loop for \((e_1, \ldots, e_n)\) in \(e : s\). Let \(\{i_1, \ldots, i_k\}\) contain the indices in \((e_1, \ldots, e_n)\) of variables not prefixed with “\(=\)”. Let \(\{j_1, \ldots, j_m\}\) contain the indices in \((e_1, \ldots, e_n)\) of constants and variables prefixed with “\(=\)”.

- Let \(\theta\) be the substitution that replaces \(e_i\) with \(\text{select}(x,i)\) for each \(i\) in \(\{i_1, \ldots, i_k\}\). Let \(x\) and \(y\) be fresh variables. 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.
if isinstance(x, Set)
  // filter out duplicate bindings.
  // y contains processed bindings.
  y = new Set
  for x in e :
    if (isTuple(x) and len(x) is n
      and select(x, j1) is e_j1 and ...
      and select(x, jm) is e_jm and bθ
      and not y.contains((select(x, i1), ..., select(x, ik)))):
      y.add((select(x, i1), ..., select(x, ik)))
    else:
      skip
  else:
    skip

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

10. Remote method invocation, i.e., invocation of a method
    on another process after that process has been started, are
    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 id, 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 de-
   clare them illegal.

3. Sets and sequences are treated as objects, because they
   are mutable. ClassName must include Set and Sequence.
   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 explicit-
   ly for a few of these methods; the others are handled
   similarly.

4. Tuples are treated as values, not as objects, because they
   are immutable.

5. Object allocation and comprehension are statements, not
   expressions, because they have side-effects. All expres-
   sions are side-effect free.

6. 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. Con-
   sequently, assignment is allowed only for instance vari-
   ables.

7. Semantically, the for loop copies the contents of a (mu-
   table) 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.

8. For brevity, among the standard arithmetic operations (+,
   -, *, etc.), we include only one representative operation
   in the abstract syntax and semantics; others are handled
   similarly.

9. The semantics below does not model real-time, so time-
   outs in await statements are simply allowed to occur
   non-deterministically.

10. We omit the concept of “site” (process location) from the
    semantics, and we omit the site argument of the construc-
    tor when creating instances of process classes, because
    process location does not affect other aspects of the se-
    mantics.

11. We do not include configure handling = all ex-
    plicitly in the program syntax, but we assume its effect
    in the semantics, since it is the default.

12. 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 but not afterward.

13. We require that all messages are tuples. This is an
    inessential restriction; it slightly simplifies the specifi-
    cation of pattern matching between messages and receive
    patterns.

14. 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 receive 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 sequences of values from domain $D$.
- $Set(D)$ contains sets of values from domain $D$.
- $D1 \xrightarrow{f} D2$ contains partial functions from $D_1$ to $D_2$.
- $\text{dom}(f)$ is the domain of a partial function $f$.

Notes.
- We require $ProcessAddress \subseteq Address$. 
Bool = \{true, false\}  
Int = ...  
Address = ...  
ProcessAddress = ...  
Tuple = Val*  
Val = Bool ∪ Int ∪ Address ∪ Tuple  
Object = (Field → Val) ∪ SetVal ∪ SeqVal  
SetVal = Set(Val)  
SeqVal = Val*  
MsgQueue = (Tuple × ProcessAddress)*  
ChannelStates = ProcessAddress × ProcessAddress → Tuple*  
HeapType = Address → ClassName  
LocalHeap = Address → Object  
Heap = ProcessAddress → LocalHeap  
State = (ProcessAddress → Statement) × HeapType × Heap × ChannelStates × (ProcessAddress → MsgQueue)

Figure 9. Semantic Domains

- For a ∈ ProcessAddress and h ∈ Heap, h(a) is the local heap of process a. For a ∈ Address and ht ∈ 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 their senders) that have arrived at the process but have not yet been received. This information is needed to express the requirement that all matching messages that have arrived at the process must be handled when execution of the process reaches a yield point.

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.

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

Expression ::=  
| Address  
| Address.Field  

Statement ::=  
| for Variable intuple Tuple: Statement

Figure 10. Extensions to the abstract syntax

C ::=  
| []  
| [Val*,C,Expr*]  
| Expr.MethodName(Val*,C,Expr*)  
| UnaryOp(C)  
| BinaryOp(C,Expr)  
| BinaryOp(Val,C)  
| isinstance(C,ClassName)  
| or(C,Expr)  
| some Pattern in C | Expr  
| C.Field = Expression  
| Address.Field = C  
| C ; Statement  
| if C: Statement else: Statement  
| for InstanceVariable in C: Statement  
| for InstanceVariable intuple Tuple: C  
| send C to Expr  
| send Val to C  
| await Expr : Stmt AnotherAwaitClause* timeout C

Figure 11. Evaluation contexts

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.

A.5 Transition Relations

The transition relation for expressions has the form e →ht,h e’, where e and e’ are expressions, ht ∈ HeapType, and h ∈ LocalHeap.

The transition relation for statements has the form σ → σ’ where σ ∈ State and σ’ ∈ 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 13–15.

Notation and auxiliary functions.
• In the transition rules, a function matches the pattern $f[x \rightarrow y]$ if $f(x)$ equals $y$.
• A function matches the pattern $f[x := y]$ if $f(x)$ equals $y$.
• $f[\pi := \rho]$ denotes the function that is the same as $f$ except that it maps $x$ to $y$.
• $f_0$ denotes the empty partial function, i.e., the partial function whose domain is the empty set.
• For a (partial) function $f$, $f \ominus a$ denotes the function that is the same as $f$ except that it has no mapping for $a$.
• Sequences are denoted with angle brackets, e.g., $(0, 1, 2) \in Int^*$. 
• $s\circ t$ is the concatenation of sequences $s$ and $t$.
• $tail(s)$ is the tail of sequence $s$, i.e., the sequence obtained by removing the first element of $s$.
• $first(s)$ is the first element of sequence $s$.
• $length(s)$ is the length of sequence $s$.
• $extends(c_1, c_2)$ holds iff class $c_1$ is a descendant of class $c_2$ in the inheritance hierarchy.

For $c \in ClassName$, 
\[
new(c, P, h) = \begin{cases} 
\{\} & \text{if } c = Set \\
\{\} & \text{if } c = Sequence \\
\text{null} & \text{if } extends(c, \text{Process)} \\
\text{null} & \text{otherwise}
\end{cases}
\]
where $newPID(P, h) = \text{any}(Int \setminus \{h(a)(a) | a \in dom(P)\})$ and $\text{any}(S)$ returns an arbitrarily selected element of set $S$.

• For $m \in MethodName$ and $c \in ClassName$, $methodDef(c, m, def)$ holds if (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 $h, h', h'' \in LocalHeap$ and $ht, ht' \in HeapType$ and $v, v' \in Val$, $isCopy(v, h, h', ht, h', ht')$ holds if (1) $v$ is a value for a process with local heap $h$ (i.e., addresses in $v$ are evaluated with respect to $h$), (2) $v'$ is a copy of $v$ for a process with local heap $h''$, i.e., $v'$ is the same as $v$ except that, instead of referencing objects in $h$, it references newly allocated copies of those objects in $h''$, and (3) $h'$ and $ht'$ are versions of $h'$ and $ht$ updated to reflect the allocation of those objects. As an exception, because process addresses are used as global identifiers, process addresses in $v$ are copied unchanged into $v'$, and new copies of process objects are not created. We give some auxiliary definitions and then a formal definition of $isCopy$. For $v \in Val$, let $addr(v, h)$ denote the set of addresses that appear in $v$ or in any objects or values reachable from $v$ with respect to local heap $h$; formally, 
\[
a \in \text{addr}(v, h) = \\
( v \in Address \land v = a)
\lor (v \in dom(h) \land h(v) \in Field \land \text{val}(\text{domain}(h(v)), a) \in addr(h(v)(f), h))
\lor (v \in dom(h) \land h(v) \in (\text{SetOfVal} \cup \text{SeqOfVal})
\lor (\exists v' \in h(v).a \in addr(v', h))
\lor (\exists v_1, \ldots, v_n \in Val. \exists i \in [1..n]. v = (v_1, \ldots, v_n)
\land a \in \text{addr}(v_i, h))
\]

For $v, v' \in Val$ and $f \in Address \rightarrow Address$, $subst(v, v', f)$ holds if $v$ is obtained from $v'$ by replacing each occurrence of an address $a$ in $dom(f)$ with $f(a)$ (informally, $f$ maps addresses of new objects in $v'$ to addresses of corresponding old objects in $v$); formally, 
\[
subst(v, v', f) = \\
( v \in \text{Bool} \cup \text{Int} \cup \text{Address} \setminus \text{dom}(f) \land v' = v)
\lor (v \in \text{dom}(f) \land f(v') = v)
\lor (\exists v_1, \ldots, v_n, v'_1, \ldots, v'_n.
\land v = (v_1, \ldots, v_n) \land v' = (v'_1, \ldots, v'_n)
\land (\forall i \in 1..n. subst(v_i, v'_i, f)))
\]

Similarly, for $o, o' \in Object$ and $f \in Address \rightarrow Address$, $subst(o, o', f)$ holds if $o$ is obtained from $o'$ by replacing each occurrence of an address $a$ in $dom(f)$ with $f(a)$. For sets $S$ and $S'$, let $S \rightarrow S'$ be the set of bijections between $S$ and $S'$. Finally, $isCopy$ is defined as follows (informally, $A$ contains the addresses of the newly allocated objects):

\[
isCopy(v, h, h', ht', ht') = \\
\exists A \subset Address \setminus ProcessAddress,
\land f \in A \rightarrow (\text{addr}(v, h) \setminus \text{ProcessAddress}).
\land A \cap \text{dom}(ht) = \emptyset \land \text{dom}(ht') = \text{dom}(ht) \cup A
\land \text{dom}(h') = \text{dom}(h) \cup A
\land (\forall a \in \text{dom}(ht). ht'(a) = ht(a))
\land (\forall a \in \text{dom}(h). h'(a) = h(a))
\land (\forall a \in A. ht'(a) = ht(f(a))
\land \text{subst}(h(a), h'(a), f))
\]

• For $m \in Val$, $a \in ProcessAddress$, $\ell \in LabelName$, $h \in LocalHeap$, 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 $matchRcvDef(m, a, \ell, h, d)$ returns the appropriately instantiated body of $d$. Specifically, if (1) either $d$ lacks an at clause, or $d$ has an at clause that includes $\ell$ (note: this implies that $\ell$ is a label name and is not the empty string), and (2) $d$ contains a receive pattern $P$ from $x$ such that there exists a substitution $\theta$ such that (2a) $m = P\theta$ and
An execution is a sequence of transitions \( \sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \cdots \) such that \( \sigma_0 \) is an initial state. The set of initial states is defined in Figure 16. 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 \( \text{skip} \), indicating that there is nothing left for the process to do), (2) get stuck (i.e., the statement associated with the process is not \( \text{skip} \), and the process has no enabled transitions) due to an unsatisfied \( \text{await} \) 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.

Figure 16. Initial states.

(2b) \( \theta(y) = h(y) \) for every variable \( y \) prefixed with “\( \_ \)” in \( P \), then, letting \( \theta \) be the substitution obtained using the first receive pattern in \( d \) for which (2) holds, \( \text{matchRcvDef}(m, a, \ell, h, d) \) returns \( s\theta[x := a] \), where \( s \) is the body of \( d \) (i.e., the statement that appears in \( d \)). Otherwise, \( \text{matchRcvDef}(m, a, \ell, d) \) returns \( \bot \).

- For \( m, \bar{m} \in \text{Val}, a \in \text{ProcessAddress}, \ell \in \text{LabelName}, c \in \text{ClassName}, \) 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 \( h \) and \( \bar{m} \) is a copy of \( m \), then \( \text{rcvAtLabel}(m, \bar{m}, a, \ell, c, h) \) returns a statement 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 sequence of receive definitions \( d \) in \( c \) (in the order that they appear in \( c \)) such that \( \text{matchRcvDef}(m, a, \ell, h, d) \) is not \( \bot \), and letting \( s_i = \text{matchRcvDef}(m, \bar{m}, a, \ell, h, d_i) \), \( \text{rcvAtLabel}(m, \bar{m}, a, \ell, c, h) \) returns \( \text{self.received.add}(\bar{m}, a); s_1; \cdots; s_n \). Otherwise, if \( \ell \) is not the empty string, and there does not exist a receive definition \( d \) in \( c \) and a label \( \ell’ \) such that \( \text{matchRcvDef}(m, a, \ell’, h, d) \) holds, then \( \text{rcvAtLabel}(m, \bar{m}, a, \ell, c, h) \) returns \( \bot \). Otherwise, \( \text{rcvAtLabel}(m, \bar{m}, a, \ell, c, h) \) returns \( \bot \).

- For \( q \in \text{MsgQueue}, \ell \in \text{LabelName}, c \in \text{ClassName}, \) and \( h \in \text{LocalHeap} \), if \( q \) contains an entry \((m, \bar{m}, a)\) such that \( \text{rcvAtLabel}(m, \bar{m}, a, \ell, c, h) \) is not \( \bot \), then, letting \((m, \bar{m}, a)\) be the first such entry in \( q \), \( \text{rcvMsg}(q, \ell, c, h) \) returns \((q’, s)\), where \( q’ \) is obtained from \( q \) by removing \((m, \bar{m}, a)\), and \( s \) is the statement returned by \( \text{rcvAtLabel}(m, \bar{m}, a, \ell, c, h) \). Otherwise, \( \text{rcvMsg}(q, \ell, c, h) \) returns \( \bot \).

A.6 Executions

An execution is a sequence of transitions \( \sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \cdots \) such that \( \sigma_0 \) is an initial state. The set of initial states is defined in Figure 16. Intuitively, \( a_p \) is the address of the
\% primitive value
\[ p \rightarrow_{ht,h} p \]

\% field access
\[ a.f \rightarrow_{ht,h} h(a)(f) \quad \text{if } a \in \text{dom}(h) \land f \in \text{dom}(h(a)) \]

\% tuple
\[ (v_1, \ldots, v_n) \rightarrow_{ht,h} (v_1, \ldots, v_n) \]

\% invoke method in user-defined class
\[ a.m(v_1, \ldots, v_n) \rightarrow_{ht,h} e[\text{self} := a, x_1 := v_1, \ldots, x_n := v_n] \]
if \( \exists c. \; ht(a) = c \land \text{methodDef}(c, m, \text{defun}(m(x_1, \ldots, x_n), e)) \)

\% invoke method in pre-defined class (representative examples)
\[ a.\text{contains}(v_1) \rightarrow_{ht,h} \text{true} \quad \text{if } \exists S. \; ht(a) = \text{Set} \land h(a) = S \land v_1 \in S \]
\[ a.\text{contains}(v_1) \rightarrow_{ht,h} \text{false} \quad \text{if } \exists S. \; ht(a) = \text{Set} \land h(a) = S \land v_1 \notin S \]

\% unary operations
\[ \text{not}(\text{true}) \rightarrow_{ht,h} \text{false} \]
\[ \text{not}(\text{false}) \rightarrow_{ht,h} \text{true} \]
\[ \text{isTuple}(v) \rightarrow_{ht,h} \text{true} \quad \text{if } v \text{ is a tuple} \]
\[ \text{isTuple}(v) \rightarrow_{ht,h} \text{false} \quad \text{if } v \text{ is not a tuple} \]
\[ \text{len}(v) \rightarrow_{ht,h} n \quad \text{if } v \text{ is a tuple with } n \text{ components} \]

\% binary operations
\[ \text{is}(v_1, v_2) \rightarrow_{ht,h} \text{true} \quad \text{if } v_1 \text{ and } v_2 \text{ are the same value} \]
\[ \text{plus}(v_1, v_2) \rightarrow_{ht,h} v_3 \quad \text{if } v_1 \in \text{Int} \land v_2 \in \text{Int} \land v_3 = v_1 + v_2 \]
\[ \text{select}(v_1, v_2) \rightarrow_{ht,h} v_3 \]
if \( v_2 \in \text{Int} \land v_2 > 0 \land (v_1 \text{ is a tuple with at least } v_2 \text{ components}) \land (v_3 \text{ is the } v_2^{\text{th}} \text{ component of } v_1) \)

\% instanceof
\[ \text{isinstance}(a, c) \rightarrow_{ht,h} \text{true} \quad \text{if } ht(a) = c \]
\[ \text{isinstance}(a, c) \rightarrow_{ht,h} \text{false} \quad \text{if } ht(a) \neq c \]

\% disjunction
\[ \text{or}(\text{true}, e) \rightarrow_{ht,h} \text{true} \]
\[ \text{or}(\text{false}, e) \rightarrow_{ht,h} e \]

\% existential quantification
\[ \text{some } x \text{ in } a \mid e \rightarrow_{ht,h} e[x := v_1] \text{ or } \cdots \text{ or } e[x := v_n] \]
if \( (ht(a) = \text{Sequence} \land h(a) = (v_1, \ldots, v_n)) \lor (ht(a) = \text{Set} \land (v_1, \ldots, v_n) \text{ is a linearization of } h(a)) \)

---

**Figure 12.** Transition relation for expressions.
% field assignment
\[(P[a \rightarrow q.a'.f = v], h, t, h[a \rightarrow ha[a' \rightarrow o]], ch, mq) \rightarrow (P[a := \text{skip}], h, t, h[a := ha[a' := o[f := v]]], ch, mq)\]
if \(\text{recvMsg}(mq(a), \ell, ht(a), h(a)) = \perp\)

% object allocation
\[(P[a \rightarrow q.a'.f = \text{new} c], h, t, h[a \rightarrow ha[a' \rightarrow o]], ch, mq) \rightarrow (P[a := \text{skip}], h, t, h[a := ha[a' := o[f := a.c]], a_c := \text{new}(c, P, h)], ch, mq)\]
if \(a_c \notin \text{dom}(ht) \land a_c \in \text{Address} \land (a_c \in \text{ProcessAddress} \iff \text{extends}(c, \text{Process}))\)

% sequential composition
\[(P[a \rightarrow \ell \text{skip} : s], h, t, h, ch, mq) \rightarrow (P[a := s], h, t, h, ch, mq)\]
if \(\text{recvMsg}(mq(a), \ell, ht(a), h(a)) = \perp\)

% conditional statement
\[(P[a \rightarrow \ell \text{if} \text{true} : s_1 \text{else} : s_2], h, t, h, ch, mq) \rightarrow (P[a := s_1], h, t, h, ch, mq)\]
if \(\text{recvMsg}(mq(a), \ell, ht(a), h(a)) = \perp\)
\[(P[a \rightarrow \ell \text{if} \text{false} : s_1 \text{else} : s_2], h, t, h, ch, mq) \rightarrow (P[a := s_2], h, t, h, ch, mq)\]
if \(\text{recvMsg}(mq(a), \ell, ht(a), h(a)) = \perp\)

% for loop
\[(P[a \rightarrow \ell \text{for} x \in a' : s], h, t, h, ch, mq) \rightarrow (P[a := \text{for} x \text{intuple} (v_1, \ldots, v_n) : s], h, t, h, ch, mq)\]
if \((ht(a) = \text{Sequence} \land h(a)(a') = (v_1, \ldots, v_n)) \lor (ht(a) = \text{Set} \land (v_1, \ldots, v_n) \text{ is a linearization of } h(a)(a'))\)
\[\land \text{recvMsg}(mq(a), \ell, ht(a), h(a)) = \perp\]

\[(P[a \rightarrow \text{for} x \text{intuple} (v_1, \ldots, v_n) : s], h, t, h, ch, mq) \rightarrow (P[a := s[x := v_1]], \text{for} x \text{intuple} (v_2, \ldots, v_n) : s], h, t, h, ch, mq)\]

\[(P[a \rightarrow \text{for} x \text{intuple} () : s], h, t, h, ch, mq) \rightarrow (P[a := \text{skip}], h, t, h, ch, mq)\]

% while loop
\[(P[a \rightarrow \ell \text{while} e : s], h, t, h, ch, mq) \rightarrow (P[a := \text{if} : (s; \text{while} e : s) \text{else} : \text{skip}], h, t, h, ch, mq)\]
if \(\text{recvMsg}(mq(a), \ell, ht(a), h(a)) = \perp\)

% invoke method in user-defined class
\[(P[a \rightarrow \ell a'.m \langle v_1, \ldots, v_n \rangle], h, t, h, ch, mq) \rightarrow (P[a := \text{self} : a, x_1 := v_1, \ldots, x_n := v_n], h, t, h, ch, mq)\]
if \(a' \in \text{dom}(ht(a))\)
\[\land ht(a') \notin \{\text{Process, Set, Sequence}\} \land \text{methodDef}(ht(a'), m, \text{def} m(x_1, \ldots, x_n) \ s)\]
\[\land \text{recvMsg}(mq(a), \ell, ht(a), h(a)) = \perp\]

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

% \text{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
\[(P[a \rightarrow \ell a'.\text{start}()], h, t, h[a \rightarrow ha[a' \rightarrow o]], ch, mq) \rightarrow (P[a := \text{skip}, a' := \text{run}()], h, a := \text{Sequence}, a_r := \text{Sequence},\]
\[h[a := ha \cup a', a' := f_o[a' \rightarrow o][\text{sent} := a_s, \text{received} := a_r, a_r := \langle \rangle, a_s := \langle \rangle]], ch, mq)\]
if \(\text{extends}(ht(a'), \text{Process}) \land (ht(a') \text{ inherits } \text{start} \text{ from } \text{Process}) \land a_r \notin \text{dom}(ht) \land a_s \notin \text{dom}(ht)\)
\[\land a_r \in \text{Address} \land \text{ProcessAddress} \land a_s \in \text{Address} \land \text{ProcessAddress} \land \text{recvMsg}(mq(a), \ell, ht(a), h(a)) = \perp\]

\textbf{Figure 13.} Transition relation for statements, part 1.
% invoke method in pre-defined class (representative examples, continued)

\[P[a \to \ell \ a'.add(v_1)], ht, h[a \to ha], ch, mq) \to (P[a := \textit{skip}, ht, h[a := ha'[a'] := ha(a') \cup \{v_1\}], ch, mq)\]

if \(a' \in \text{dom}(ha) \land ht(a') = \text{Set} \land \text{rcvMsg}(mq(a), \ell, ht(a), h(a)) = \bot\)

\[P[a \to \ell \ a'.append(v_1)], ht, h[a \to ha], ch, mq) \to (P[a := \textit{skip}, ht, h[a := ha'[a'] := ha(a') \& ha(v_1)], ch, mq)\]

if \(a' \in \text{dom}(ha) \land ht(a') = \text{Sequence} \land ht(v_1) = \text{Sequence} \land \text{rcvMsg}(mq(a), \ell, ht(a), h(a)) = \bot\)

% send to a process. create copies of the message for the sender’s send history and the receiver.

\[P[a \to \ell \text{send} v \to a'], ht, h[a \to ha], a', ch, mq) \to (P[a := \textit{skip}, ht_2, h[a := ha_1[a_a] := ha(a_a)@(v_1, a']], a' := ha)([a', a'] := (v')@ch((a, a'))], ch, mq)\]

if \(a' \in \text{ProcessAddress} \land a_a = ha(a)(\textit{sent}) \land \text{isCopy}(v, ha, ha, ht_1, ha_1, ht_2) \land \text{rcvMsg}(mq(a), \ell, ht(a), h(a)) = \bot\)

% send to a set of processes

\[P[a \to \ell \text{send} v \to a'], ht, h[a \to ha], ch, mq) \to (P[a := \textit{for} x \in a': \text{send} v \to x], ht, h[a := ha[a_a] := ha(a_a)@(v_1, a')], ch, mq)\]

if \(ht(a') = \text{Set} \land a_a = ha(a)(\textit{sent}) \land (x \text{ is a fresh variable}) \land \text{rcvMsg}(mq(a), \ell, ht(a), h(a)) = \bot\)

% message reordering

\(P, ht, h, ch[(a, a') \to q], mq) \to (P, ht, h, ch[(a, a') := q'], mq)\)

if \(\text{(channel order is unordered in the program configuration)} \land (q' \text{ is a permutation of } q)\)

% message loss

\(P, ht, h, ch[(a, a') \to q], mq) \to (P, ht, h, ch[(a, a') := q'], mq)\)

if \(\text{(channel reliability is unreliable in the program configuration)} \land (q' \text{ is a subsequence of } q)\)

% receive a message

\[P[a \to \ell \text{set} s], ht, h, ch, mq[a \to q]) \to (P[a := s[\textit{self} := a]; \ell s], ht, h, ch, mq[a := q'])\]

if \(\text{rcvMsg}(mq(a), \ell, ht(a), h(a)) = (q', s')\)

% arrival of message at process (message moves from channel to message queue)

% append tuple \((m, m', a)\) to message queue. \(m'\) is a copy of message \(m\) for storage in receive history.

\(P, ht, h, a'[a' \to q], ch[(a, a') \to q], mq)\)

\(\to (P, ht', [a'[a' \to ha'], ch[(a, a') := tail(q)], mq'[a' := mq(a')@(first(q), m', a)])\)

if \(\text{length}(q) > 0 \land \text{isCopy(first(q), ha, ha, ha, ha', ha', ht')}\)

% await without timeout clause

\(P[a \to \ell \text{wait} e_1; s_1 \lor \cdots \lor e_n; s_n], ht, h, ch, mq) \to (P[a := s_i], ht, h, ch, mq)\)

if \(i \in [1..n] \land e_i \to_{ht,h(a)} \text{true} \land \text{rcvMsg}(mq(a), \ell, ht(a), h(a)) = \bot\)

% await with timeout clause, terminated by true condition

\(P[a \to \ell \text{wait} e_1; s_1 \lor \cdots \lor e_n; s_n \text{timeout} v:s], ht, h, ch, mq) \to (P[a := s_i], ht, h, ch, mq)\)

if \(i \in [1..n] \land e_i \to_{ht,h(a)} \text{true} \land \text{rcvMsg}(mq(a), \ell, ht(a), h(a)) = \bot\)

% await with timeout clause, terminated by timeout (occurs non-deterministically)

\(P[a \to \ell \text{wait} e_1; s_1 \lor \cdots \lor e_n; s_n \text{timeout} v:s], ht, h, ch, mq) \to (P[a := s], ht, h, ch, mq)\)

if \(e_1 \to_{ht,h(a)} \text{false} \land \cdots \land e_n \to_{ht,h(a)} \text{false} \land \text{rcvMsg}(mq(a), \ell, ht(a), h(a)) = \bot\)

\[\text{Figure 14. Transition relation for statements, part 2.}\]
% context rule for expressions

\[ e \rightarrow_{ht,h(a)} e' \]

\[(P[a \rightarrow C[e]], ht, h, ch, mq) \rightarrow (P[a := C[e']], ht, h, ch, mq)\]

% context rule for statements

\[(P[a \rightarrow s], ht, h, ch, mq) \rightarrow (P[a := s'], ht', h', ch', mq')\]

\[(P[a \rightarrow C[s]], ht, h, ch, mq) \rightarrow (P[a := C[s']], ht', h', ch', mq')\]

Figure 15. Transition relation for statements, part 3.