DistAlgo Language Description

Yanhong A. Liu, Bo Lin, and Scott Stoller

liu@cs.stonybrook.edu, bolin@cs.stonybrook.edu, stoller@cs.stonybrook.edu

Revised March 24, 2017

DistAlgo is a language for distributed algorithms. We describe DistAlgo language constructs as extensions to conventional object-oriented programming languages, including a syntax for extensions to Python.

There are four components conceptually: (1) distributed processes and sending messages, (2) control flows and receiving messages, (3) high-level queries of message histories, and (4) configurations.

High-level queries are not specific to distributed algorithms, but using them over message histories is particularly helpful for expressing and understanding distributed algorithms at a high level. Some conventional programming languages, such as Python, support high-level queries to some extent, but DistAlgo query constructs are more declarative, especially with the support of tuple patterns for messages.

1 Distributed processes and sending messages

1.1 Process definition

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

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

The syntax of process definition could be made simpler and clearer:

```
process p:
  process_body
```

but it would make `process` a keyword, which is usually a reserved word, whereas `process` as a class name is not reserved and can be defined or redefined to be anything else.

⟶ in Python syntax:

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

A special method `setup` may be defined in *process_body* for initially setting up data in the process before the process's execution starts. For each parameter v of `setup`, a process field named v is defined automatically and assigned the value of parameter v; additional fields can be defined explicitly in the method body of `setup`.

A special method `run()` may be defined in *process_body* for carrying out the main flow of execution.

A special variable `self` refers to the process itself. Fields of the process may be defined by including the field name as a parameter of method `setup`, or by explicitly prefixing the field name with `self` in an assignment to the field. References to fields of the process do not need to be prefixed with `self`. References to methods of the process do not need to be prefixed with `self` either. Also, method definitions implicitly include parameter `self`.

1.2 Process creation

Process creation consists of statements for creating, setting up, and starting processes.

A process creation statement is of the following form. It creates n new processes of type p at each node in the value of *node_exp*, and returns the resulting process or set of processes.

n `new` *p* `at` *node_exp*

Expression *node_exp* evaluates to a node or a set of nodes, specifying where the new processes will be created. A node is a running DistAlgo program on a machine. A node is identified by a string of the form `name@host`, where `name` can be specified on the command line when starting the node, and `host` is the host name of the machine running the node; `@host` can be omitted if the node is running on the local machine. All nodes communicating with each other must have the same cookie, which can be specified on the command line when starting the node. The number n and the `at` clause are optional; the defaults are 1 and local node, respectively. When both are omitted, a single process is created and returned; otherwise, a set of processes is created and returned.

⟶ **in Python syntax:**

`new(`*p*`, num = `*n*`, at = `*node_exp*`)`

A process setup statement is of the following form. It sets up the process or set of processes that is the value of expression *pexp*, using method `setup` of the process or processes with the values of argument expressions *args*. If the values of *args* are available when the process or processes are created at a call to `new`, the call to `setup` can be omitted by inserting tuple `(`*args*`)` after *p* in the call to `new`.

pexp`.setup(`*args*`)`

⟶ **in Python syntax:**

`setup(`*pexp*`, (`*args*`))` Note: You must add a trailing comma if *args* is a single argument.

A process start statement is of the following form. It starts the execution of the method `run` of the process or set of processes that is the value of expression *pexp*.

pexp`.start()`

⟶ **in Python syntax:**

`start(`*pexp*`)`

1.3 Sending messages

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

`send` *mexp* `to` *pexp*

⟶ **in Python syntax:**

`send(`*mexp*`, to = `*pexp*`)`

2 Control flows and receiving messages

Received messages can be handled both asynchronously, using yield points and `receive` definitions, and synchronously, using `await` statements.

2.1 Yield points

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 yields to handling of un-handled messages, if any, and resumes afterwards.

`--` l`:`

⟶ in Python syntax:

`--` l

which is a statement in Python, where l is any valid Python identifier.

2.2 Handling received messages

A handler definition, also called a `receive` definition, is of the following form. It handles, at yield points labeled l_1, ..., l_j, un-handled messages that match *mexp* from *pexp*, where *mexp* and *pexp* are parts of a pattern, to be described. The `from` and `at` clauses are optional; the defaults are any process and all yield points. The *handler_body* is a sequence of statements to be executed for the matched messages.

`receive` *mexp* `from` *pexp* `at` l_1`, ..., `l_j`:`
  handler_body

⟶ in Python syntax:

`def receive(msg =` *mexp*`, from_ =` *pexp*`, at = (`l_1`, ..., `l_j`)):`
  handler_body

where `_` is added after `from` because `from` is a reserved word in Python.

2.3 Synchronization

A simple `await` statement is of the following form. It waits for the value of Boolean-valued expression *bexp* to become true.

`await` *bexp*

⟶ in Python syntax:

`await(`*bexp*`)`

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

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

⟶ in Python syntax:

```
if await(bexp_1): stmt_1
elif ...
elif bexp_k: stmt_k
elif timeout(t): stmt
```

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

3 High-level queries of message histories

High-level queries can be used over message histories, and patterns can be used for matching the messages.

3.1 Message histories

Histories of message sent and received by a process are kept in variables `sent` and `received`, respectively. Sequence `sent` is updated at each `send` statement, by adding each message sent to a process. Sequence `received` is updated at the next yield point if there are un-handled messages, by adding un-handled messages before any matching receive handler executes.

The following two expressions are equivalent and return true iff a message that matches *mexp* to *pexp* is in `sent`. The `to` clause is optional; the default is any process.

```
sent mexp to pexp
mexp to pexp in sent
```

⟶ in Python syntax:

```
sent(mexp, to = pexp)
(mexp, pexp) in sent
```

Similarly, the following expressions use `received`.

```
received mexp from pexp
mexp from pexp in received
```

⟶ in Python syntax:

```
received(mexp, from_ = pexp)
(mexp, pexp) in received
```

3.2 Patterns

A pattern can be used to match a message, in `sent` and `received`, and by a `receive` definition. A constant value, such as `'ack'`, or a previously bound variable, indicated with prefix `=`, in the pattern must match the corresponding components of the message. An underscore (`_`) matches anything. Previously unbound variables in a pattern are bound to the corresponding components in the matched message. In general, any data construction expression can be used as a pattern; we use only tuple patterns because messages are by convention tuples.

For example, `received ('ack',=n,_) from a` matches every triple in `received` whose first two components are `'ack'` and the value of `n`, and binds `a` to the sender.

Notation `=x` means a value that is equal to the value of variable `x`; it is equivalent to using a fresh variable `y` in its place and adding a test `y=x` before `x` is used. This notation can generalize: one can add as prefix any binary operator that is a symbol not allowed in identifiers, uses the parameter value as the right operand, and returns a Boolean value. For example, `>x` means a value that is greater than the value of parameter `x`.

⟶ in Python syntax: `_` is used in place of `=` to indicate previously bound variables. This forbids the use of variable names that start with `_` in the query. Other binary operators cannot be used as prefix.

3.3 Comprehension, aggregation, and quantification

A query is over sets or sequences. It can be a comprehension, aggregation, or quantification, plus a set of *parameters*—variables whose values are bound before the query. For a query to be well-formed, every variable in it must be *reachable* from a parameter—be a parameter or be the left-side variable of a membership clause whose right-side variables are reachable.

To indicate that a variable `x` on the left side of a membership clause is a parameter, add prefix `=` to `x`; this is only needed for the first occurrence of such a variable. Use of `=` to indicate parameters is consistent with use of `=` in patterns.

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

{*exp*: v_1 `in` $sexp_1$, ..., v_k `in` $sexp_k$, *bexp*}

⟶ in Python syntax:

`setof(`*exp*`,` v_1 `in` $sexp_1$`, ...,` v_k `in` $sexp_k$`,` *bexp*`)`

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

agg *sexp*
agg (*exp*: v_1 `in` $sexp_1$, ..., v_k `in` $sexp_k$, *bexp*)

⟶ in Python syntax:

agg`(`*sexp*`)`
agg`of(`*exp*`,` v_1 `in` $sexp_1$`, ...,` v_k `in` $sexp_k$`,` *bexp*`)`

where `len` is used in place of `count`.

Quantification. A quantification is a query of one of the following two forms, called existential and universal quantifications, respectively. Given values of parameters, the quantification returns `true` iff for some or all, respectively, combinations of values of variables that satisfy all membership clauses v_i `in` $sexp_i$, condition *bexp* evaluates to `true`. When an existential quantification returns `true`, all variables in the query are also bound to a combination of values, called a witness, that satisfy all the membership clauses and condition *bexp*.

`some` v_1 `in` $sexp_1$`, ...,` v_k `in` $sexp_k$ `has` *bexp*
`each` v_1 `in` $sexp_1$`, ...,` v_k `in` $sexp_k$ `has` *bexp*

⟶ in Python syntax:

`some(`v_1 `in` $sexp_1$`, ...,` v_k `in` $sexp_k$`, has =` *bexp*`)`
`each(`v_1 `in` $sexp_1$`, ...,` v_k `in` $sexp_k$`, has =` *bexp*`)`

Other forms for membership and condition. In all of comprehension, aggregation, and quantification, in membership clause v_i `in` $sexp_i$, if $sexp_i$ is a variable s_i, then clause v_i `in` s_i can also be written as $s_i(v_i)$. Also, a tuple pattern may be used in place of variable v_i. When condition *bexp* is `true`, the clause with *bexp* can be omitted.

⟶ **in Python syntax:** Only for $sexp_i$ being variable `sent` or `received` can clause v_i `in received` or v_i `in sent` be written as `received(`v_i`)` or `sent(`v_i`)`, respectively.

4 Configurations

4.1 Channel types

The following statement configures all channels to be first-in-first-out (FIFO). Other options for `channel` include `reliable` and `{reliable, fifo}`. When these options are specified, TCP is used for process communication; otherwise, UDP is used.

```
configure channel = fifo
```

⟶ **in Python syntax:**

```
config(channel = 'fifo')
```

Channels can also be configured separately for messages from certain types of processes to certain types of processes, by adding clauses `from` *ps* and `to` *qs*, or arguments `from_ =` *ps* and `to =` *qs* in Python syntax, where *ps* and *qs* can be a type of processes or a set of types of processes. Each of these clauses is optional; the default is all types of processes.

4.2 Message handling

The following statement configures the system to handle all un-handled messages at each yield point; this is the default. Other options for `handling` include `one`.

```
configure handling = all
```

⟶ **in Python syntax:**

```
config(handling = 'all')
```

4.3 Logical clocks

The following statement configures the system to use Lamport clock. Other options for `clock` include `vector`; it is currently not supported.

```
configure clock = Lamport
```

⟶ **in Python syntax:**

```
config(clock = 'Lamport')
```

A call `logical_time()` returns the current value of the logical clock.

4.4 Overall

A DistAlgo program is written in files named with extension `.da`. It consists of a set of process definitions, a method `main`, and possibly other, conventional program parts. Method `main` specifies the configurations and creates, sets up, and starts a set of processes.

DistAlgo language constructs can be used in process definitions and method `main` and are implemented according to the semantics described; other, conventional program parts are implemented according to their conventional semantics.

5 Other useful functions in Python

5.1 Logging output

The following method prints the values of expressions exp_1 through exp_k in their `str()` representation, separated by the value of *str_exp* and prefixed with system timestamp, process id, and the specified integer level l, to the log of the node that runs the current DistAlgo process; the printing is done only if level l is greater or equal to the default logging level or the level specified on the command line when starting the node. The log defaults to console, but can be a file specified on the command line when starting the node.

`output(`exp_1`, ..., `exp_k`, sep = `*str_exp*`, level = `l`)`

Argument `sep` is optional and defaults to the empty space. Argument `level` is optional and defaults to `logging.INFO`, corresponding to value 20, in the Python logging module; see `https://docs.python.org/3/library/logging.html#levels` for a list of predefined level names.

5.2 Importing modules

A DistAlgo module *module* is a DistAlgo file named *module*`.da`. DistAlgo files can be imported just as Python files.

For example, the following statement takes DistAlgo module *module* in file *module*`.da`, compiles it if an up-to-date compiled file does not already exist, and assigns to m the resulting module object if successful or raises `ImportError` otherwise.

`import` *module* `as` m