DistAlgo Language Description

Yanhong A. Liu, Bo Lin, and Scott Stoller

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

September 26, 2014

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 for this purpose only 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 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()` must be defined in *process_body* for carrying out the main flow of execution.

A special variable `self` refers to the current process. A special field `id` holds the id of the process. All other fields of the process must be defined in method `setup`, by including the field name as a parameter of `setup`, or by explicitly prefixing the field name with `self` in an assignment to the field. Other 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, and assigns the single new process or set of new processes to variable v. Expression *node_exp* evaluates to a node or a set of nodes, for where the new processes will be created.[1] The number `n` and clause `at` are optional; the defaults are 1 and local node, respectively.

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

⟶ in Python syntax:

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

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))
```

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] The `at` clause is not supported in this version.

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

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 may yield to handling of received messages.

```
-- l:
```

⟶ in Python syntax:

```
-- l
```

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

2.2 Handling messages received

Handling messages received can be done using handler definitions and message history variables.

A handler definition is of the following form, It handles, at yield points labeled l_1, ..., l_j, any un-handled message that matches the value of *mexp* and is sent from the values of *pexp*. 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 l1, ..., lj:
  handler_body
```

We could use the noun `handler` in place of `receive`, but handlers are not named and called with their names; instead, yield points are named, and handlers are executed at the specified yield points.

⟶ in Python syntax:

```
def receive(mexp, from_ = pexp, at = (l1, ..., lj)):
  handler_body
```

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

Message histories, i.e., the sequences of messages received and sent, in variables `received` and `sent`, respectively, can be used in expressions.

In particular, the following two equivalent expressions return true iff a message that matches the value *mexp* and is sent from the value of *pexp* is in `received`. The `from` clause is optional; the default is any process.

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

⟶ in Python syntax:

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

Similarly, the following expressions use `sent`.

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

⟶ in Python syntax:

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

2.3 Synchronization

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 *bexp* becomes true. It is a short hand for `await` *bexp*`: pass` in a general, nondeterministic await statement.

```
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 bexp1: stmt1
or ...
or bexpk: stmtk
timeout t: stmt
```

⟶ **in Python syntax:**

```
if await(bexp1): stmt1
elif ...
elif bexpk: stmtk
elif timeout(t): 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.

3 High-level queries of message histories

3.1 Comprehensions

A comprehension is a query of the following form 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 variable is reachable. Given values of parameters, the query returns the set of values of *exp* for all values of variables that satisfy all membership clauses v_i `in` exp_i and condition *bexp*. When $sexp_i$ is a variable s_i, expression $s_i(v_i)$ can be used in place of v_i `in` s_i. When *bexp* is `true`, `,` *bexp* can be omitted.

```
{exp: v1 in sexp1, ..., vk in sexpk, bexp}
```

To indicate any variable `x` on the left side of a membership clause to be a parameters, add prefix `=` to `x`. Notation `=x` means a value 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*. 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:**

```
setof(exp, v1 in sexp1, ..., vk in sexpk, bexp)
```

where `_` is used in place of `=` to indicate parameters. This forbids the use of variable names that start with `_` in the query.

3.2 Aggregates

An aggregate is a query of the following form, where *agg_op* is an aggregate operator, including `count`, `sum`, `min`, and `max`. The query returns the value of applying *agg_op* to the set value of the comprehension expression *comprehension_exp*.

```
agg_op comprehension_exp
```

⟶ in Python syntax:

```
agg_op(comprehension_exp)
```

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

3.3 Quantifications

A quantification is a query of one of the following two forms plus a set of parameters. The two forms are called existential and universal quantifications, respectively. Given values of parameters, the query returns `true` iff for some or all, respectively, values of the variables that satisfy all membership clauses, *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 v1 in sexp1, ..., vk in sexpk has bexp
each v1 in sexp1, ..., vk in sexpk has bexp
```

Parameters are indicated as for comprehensions. Also as for comprehensions, when $sexp_i$ is a variable s_i, expression $s_i(v_i)$ can be used in place of v_i `in` s_i. When *bexp* is `true`, the `has` clause can be omitted.

⟶ in Python syntax:

```
some(v1 in sexp1, ..., vk in sexpk, has = bexp)
each(v1 in sexp1, ..., vk in sexpk, has = bexp)
```

where prefix `_` or a `params` clause is used to indicate parameters, as for comprehensions.

3.4 Patterns

In the clauses v_1 `in` $sexp_1$ `, ..., ` v_k `in` $sexp_k$ in all of comprehensions, aggregates, and quantifications, a tuple expression $texp_i$, called a tuple pattern, may occur in place of variable v_i. Variables in $texp_i$ are bound to the corresponding components in elements of the value of $sexp_1$. The underscore (`_`) is used as a wild card that can be bound to anything. In general, any data construction expression can be used as a pattern; we use only tuple patterns because messages are by convention tuples.

4 Configurations

4.1 Channel types

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

```
configure channel = fifo
```

$\longrightarrow$ **in Python syntax:**

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

Channels can also be configured separately for communication among any set of processes, say *ps*, by adding a clause `among` *ps*, or `among =` *ps* in Python syntax.

4.2 Message handling

The following statements configures the system to handle all messages received at each yield point; this is the default.

```
configure handling = all
```

$\longrightarrow$ **in Python syntax:**

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

4.3 Logical clock

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

```
configure clock = Lamport
```

$\longrightarrow$ **in Python syntax:**

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

Function `logical_clock()` 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 value of string expression *str_exp*, prefixed with system timestamp, process id, and the specified level l, to the log of the current run on 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 DistAlgo run. The log is default to console, but can be a file specified on the command line when starting the DistAlgo run.

```
output(message = str_exp, level = l)
```

Argument `level` is optional and is default 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

The following statement is equivalent to Python statement `import` *module* `as` m. It takes DistAlgo module *module*, which must end in a DistAlgo program file name excluding extension `.da`, compiles the program file 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.

```
m = import_da(module)
```