```
    add(3, 2), result(5), add(1, 2), add(-1, 1), result(0), ...
```
that start with an `add` action, in which every `result` action returns the sum computed by the last `add` action, and in which every pair of `result` actions must be separated by one or more `add` actions.

```
automaton Channel(M, Index: type, i, j: Index)
  signature
    input  send(m: M, const i, const j)
    output receive(m: M, const i, const j)
  states
    buffer: Seq[M] := {}
  transitions
    input send(m, i, j)
      eff buffer := buffer ⊢ m
    output receive(m, i, j)
      pre buffer ≠ {} ∧ m = head(buffer)
      eff buffer := tail(buffer)
```

Figure 5: IOA description of a reliable communication channel

Another simple automaton, `Channel`, is shown in Figure 5. This automaton represents a reliable communication channel, as illustrated in Figure 2, which neither loses nor reorders messages in transit. The automaton is parameterized by the type `M` of messages that can be in transit on the channel, by the type `Index` of process indices, and by two values, `i` and `j`, which represent the indices of processes that use the channel for communication. The signature consists of input actions, `send(m, i, j)`, and output actions, `receive(m, i, j)`, one for each value of `m`. The keyword **const** in the signature indicates that the values of `i` and `j` in these actions are fixed by the values of the automaton's parameters.

The state of the automaton `Channel` consists of a `buffer`, which is a sequence of messages (i.e., an element of type `Seq[M]`) initialized to the empty sequence `{}`. Section 8.4 describes the type constructor `Seq` and operators on sequences such as `{}`, ⊢, `head`, and `tail`.

The input action `send(m, i, j)` has the effect of appending `m` to `buffer` (here, ⊢ is the append operator). The output action `receive(m, i, j)` is enabled when `buffer` is not empty and has the message `m` at its head. The effect of this action is to remove the head element from `buffer`.

The rest of Part I shows in more detail how IOA can be used to describe I/O automata.

3 Data types in IOA descriptions

IOA enables users to define the actions and states of I/O automata abstractly, using mathematical notations for sets, sequences, etc., without having to provide concrete representations for these abstractions. Some mathematical notations are built into IOA; others can be defined by the user.

The data types `Bool`, `Int`, `Nat`, `Real`, `Char`, and `String` can appear in IOA descriptions without explicit declarations. Section 7 describes the operators available for each of these types.

Compound data types can be constructed using the following type constructors and used without explicit declarations. Section 8 describes the operators available for types constructed in any of these fashions.

- `Array[I, E]` is an array of elements of type `E` indexed by elements of type `I`.
- `Map[D, R]` is a finite partial mapping of elements of a domain type `D` to elements of a range type `R`. Mappings differ from arrays in that their domains are always finite, and in that they may not be totally defined.

- `Seq[E]` is a finite sequence of elements of type `E`.
- `Set[E]` is a finite set of elements of type `E`.
- `Mset[E]` is a finite multiset of elements of type `E`.

In this tutorial, we describe operators on the built-in data types informally when they first appear in an example.

Users can define additional data types, as well as redefine built-in types, in one of two ways. First, they can explicitly declare enumeration, tuple, and union types analogous to those found in many common programming languages. For example,

```
type Color = enumeration of red, white, blue
type Msg   = tuple of source, dest: Process, contents: String
type Fig   = union of sq: Square, circ: Circle
```

Section 9.8 describes the operators available for each of these types. Second, users can refer to an auxiliary specification that defines the syntax and semantics of a data type, as in

```
axioms Queue for Q[__]   % Supplies axioms for Q[Int], Q[Set[Nat]], ...
axioms Peano for Nat     % Overrides built-in axioms for Nat
axioms Graph(V, E)       % Supplies axioms for graphs
```

These auxiliary specifications are written in the *Larch Shared Language* (*LSL*); see Sections 9 and 10.

In this report, some operators are displayed using mathematical symbols that do not appear on the standard keyboard. The following tables show the input conventions for entering these symbols. The standard meanings of the logical operators are built into LSL and IOA. The meanings of the datatype operators are defined by the LSL specifications for the built-in datatypes in Section 9.

Logical Operator			Datatype Operator		
Symbol	Meaning	Input	Symbol	Meaning	Input
$\forall$	For all	`\A`	$\leq$	Less than or equal	`<=`
$\exists$	There exists	`\E`	$\geq$	Greater than or equal	`>=`
$\neg$	Not	`~`	$\in$	Member of	`\in`
$\neq$	Not equals	`~=`	$\notin$	Not a member of	`\notin`
$\wedge$	And	`/\`	$\subset$	Proper subset of	`\subset`
$\vee$	Or	`\/`	$\subseteq$	Subset of	`\subseteq`
$\Rightarrow$	Implies	`=>`	$\supset$	Proper superset of	`\supset`
$\Leftrightarrow$	If and only if	`<=>`	$\supseteq$	Superset of	`\supseteq`
			$\vdash$	Append element	`|-`
			$\dashv$	Prepend element	`-|`

4 IOA descriptions for primitive automata

Primitive automata (i.e., automata without subcomponents) are described by specifying their names, action signatures, state variables, transition relations, and task partitions. All but the last of these elements must be present in every primitive automaton description.

4.1 Automaton names and parameters

The first line of an automaton description consists of the keyword **automaton** followed by the name of the automaton (see Figures 4 and 5). As illustrated in Figure 5, the name may be followed by a list of formal parameters enclosed within parentheses. Each parameter consists of an identifier with its associated type (or, as in Figure 5, with the keyword **type** to indicate that the identifier names a type rather than an element of a type).[4]

4.2 Action signatures

The signature for an automaton is declared in IOA using the keyword **signature** followed by lists of entries describing the automaton's input, internal, and output actions. Each entry contains a name and an optional list of parameters enclosed in parentheses. Each parameter consists of an identifier with its associated type, or of an expression following the keyword **const**; entries cannot have **type** parameters. Each entry in the signature denotes a set of actions, one for each assignment of values to its non-**const** parameters.

It is possible to place constraints on the values of the parameters for an entry in the signature using the keyword **where** followed by a predicate, that is, by a boolean-valued expression. Such constraints restrict the set of actions denoted by the entry. For example, the signature

```
  signature
    input  add(i, j: Int) where i > 0 ∧ j > 0
    output result(k: Int) where k > 1
```

could have been used for the automaton `Adder` to restrict the values of the input parameters to positive integers and the value of the output parameter to integers greater than 1.

4.3 State variables

As in the above examples, state variables are declared using the keyword **states** followed by a comma-separated list of state variables and their types. State variables can be initialized using the assignment operator := followed by an expression or by a nondeterministic choice. The order in which state variables are declared makes no difference: state variables are initialized simultaneously, and the initialization given for one state variable cannot refer to the value of any other state variable.

A nondeterministic choice, indicated by the keyword **choose** following the assignment operator :=, selects an arbitrary value for the named variable that satisfies the predicate following the keyword **where**. When a nondeterministic choice is used to initialize a state variable, there must be some value of the named variable that satisfies this predicate. If this predicate is true for all values of the named variable, then the effect is the same as if no initial value had been specified for the state variable.

```
automaton Choice
  signature
    output result(i: Int)
  states
     num:  Int  := choose n where 1 ≤ n ∧ n ≤ 3,
     done: Bool := false
  transitions
    output result(i)
      pre ¬done ∧ i = num
      eff done := true
```

Figure 6: Example of nondeterministic choice of initial value for state variable

[4]Later versions of IOA may also allow us to parameterize automata by operations (e.g., ordering relations) on a data type.

For example, in the automaton `Choice` (Figure 6), the state variable `num` is initialized nondeterministically to some value of the variable `n` that satisfies the predicate $1 \leq n \wedge n \leq 3$, i.e., to one of the values 1, 2, or 3 (the value of `n` must be an integer because it is constrained to have the same type, `Int`, as the variable `num` to which it will be assigned). The automaton `Choice` can return the selected value at most once in an output action.

It is also possible to constrain the initial values of all state variables taken together, whether or not initial values are assigned to any individual state variable. This can be done using the construct **so that** followed by a predicate (involving state variables and automaton parameters), as illustrated by the definition of the automaton `Shuffle` in Figure 7.[5] Here, the initial values of the variable `cut` and the array `name` of strings are constrained so that `name[1]`, ..., `name[52]` are sorted in two pieces, each in increasing order, with the piece after the `cut` containing smaller elements than the piece before the `cut`. Note that the scope of the **so that** clause is the entire set of state variable declarations.

```
type cardIndex = enumeration of 1, 2, 3, ..., 52

automaton Shuffle
  signature
    internal swap(i, j: cardIndex)
    output   deal(a: Array[cardIndex, String])
  states
    dealt: Bool := false,
    name:  Array[cardIndex, String],
    cut:   cardIndex,
    temp:  String
    so that ∀ i: cardIndex (i ≠ 52 ∧ i ≠ cut ⇒ name[i] < name[succ(i)])
              ∧ name[52] < name[1]
  transitions
    internal swap(i, j)
      pre ¬dealt
      eff temp := name[i];
          name[i] := name[j];
          name[j] := temp
    output deal(a)
      pre ¬dealt ∧ a = name
      eff dealt := true
```

Figure 7: Example of a constraint on initial values for state variables

In Figure 7, values of type `Array[cardIndex, String]` are arrays indexed by elements of type `cardIndex` and containing elements of type `String` (see Section 8.1). The `swap` actions transpose pairs of strings, until a `deal` action announces the contents of the array; then no further actions occur. Note that the constraint following **so that** limits only the initial values of the state variables, not their subsequent values.

When the type of a state variable is an `Array` or a **tuple** (Section 9.8), IOA also treats the elements of the array or the fields in the tuple as state variables, to which values can be assigned without affecting the values of the other elements in the array or fields in the tuple.

[5]At present, users must expand the ... in the definition of the type `cardIndex` by hand; IOA will eventually provide more convenient notations for integer subranges.

4.4 Transition relations

Transitions for the actions in an automaton's signature are defined following the keyword **transitions.** A transition definition consists of an action type (i.e., **input**, **internal**, or **output**), an action name with optional parameters and an optional **where** clause, an optional list of additional "**choose** parameters," an optional precondition, and an optional effect.

4.4.1 Transition parameters

The parameters accompanying an action name in a transition definition must match those accompanying the name in the automaton's signature, both in number and in type. However, parameters take a simpler form in a transition definition than they do in the signature. The simplest way to construct the parameter list for an action name in a transition definition is to erase the keyword **const** and the type modifiers from the parameter list in the signature; thus, in Figure 5,

```
  input send(m: M, const i, const j)
```

in the signature of `Channel` is shortened to **input** `send(m, i, j)` in the transition definition. See Section 15.3 for the actual set of rules.

More than one transition definition can be given for an entry in an automaton's signature. For example, the transition definition for the `swap` actions in the `Shuffle` automaton (Figure 7) can be split into two components:

```
    internal swap(i, j) where i ≠ j
      pre ¬dealt
      eff temp := name[i];
          name[i] := name[j];
          name[j] := temp

    internal swap(i, i)
      pre ¬dealt
```

The second of these two transition definitions does not change the state, because it has no **eff** clause.

4.4.2 Preconditions

A precondition can be defined for a transition of an output or internal action using the keyword **pre** followed by a predicate, that is, by a boolean-valued expression. Preconditions cannot be defined for transitions of input actions. All variables in the precondition must be parameters of the automaton, be state variables, appear in the parameter list for the transition definition, be **choose** parameters, or be quantified explicitly in the precondition. If no precondition is given, it is assumed to be `true`.

An action is said to be *enabled* in a state if the precondition for its transition definition is true in that state for some values of the **choose** parameters. Input actions, whose transitions have no preconditions, are always enabled.

4.4.3 Effects

The effect of a transition, if any, is defined following the keyword **eff.** This effect is generally defined in terms of a (possibly nondeterministic) program that assigns new values to state variables. The amount of nondeterminism in a transition can be limited by a predicate relating the values of state variables in the post-state (i.e., in the state after the transition has occurred) to each other and to their values in the pre-state (i.e., in the state before the transition occurs).

If the effect is missing, then the transition has none; i.e., it leaves the state unchanged.

Using programs to specify effects A program is a list of statements, separated by semicolons. Statements in a program are executed sequentially. There are three kinds of statements:

- assignment statements,
- conditional statements, and
- **for** statements.

Assignment statements An assignment statement changes the value of a state variable. The statement consists of a state variable followed by the assignment operator := and either an expression or a nondeterministic choice (indicated by the keyword **choose**). (As noted in Section 4.3, the elements in an array used as a state variable, or the fields in a tuple used as a state variable, are themselves considered as separate state variables and can appear on the left side of the assignment operator.)

The expression or nondeterministic choice in an assignment statement must have the same type as the state variable. The value of the expression is defined mathematically, rather than computationally, in the state before the assignment statement is executed. The value of the expression then becomes the value of the state variable in the state after the assignment statement is executed. Execution of an assignment statement does not have side-effects; i.e., it does not change the value of any state variable other than that on the left side of the assignment operator.

```
axioms Subsequence for Seq[__]

automaton LossyChannel(M: type)
  signature
    input  send(m: M),
           crash
    output receive(m: M)
  states
    buffer: Seq[M] := {}
  transitions
    input send(m)
      eff buffer := buffer ⊢ m
    input crash
      eff buffer := choose b where b ≼ buffer
    output receive(m)
      pre buffer ≠ {} ∧ m = head(buffer)
      eff buffer := tail(buffer)
```

Figure 8: IOA description of a lossy communication channel

The definition of the `crash` action in the `LossyChannel` automaton (Figure 8) illustrates the use of the **choose ... where** construct to constrain the new value of the state variable `buffer` to be a nondeterministically chosen subsequence of the old value. `LossyChannel` is a modification of the reliable communication channel (Figure 5) in which the additional input action `crash` may cause the sequence `buffer` to lose messages (but not to reorder them).

The **axioms** statement at the beginning of Figure 8 identifies an auxiliary specification (Figure 9), which overrides the default axioms for the built-in type constructor `Seq[E]` for the sequence data type (see Section 8.4) to add a definition for the subsequence relation $\preceq$ appearing in the definition of transitions for the `crash` action. Because this relation is not one of the built-in operators provided by IOA for the sequence data type, we must supply a specification to define its properties, namely, that a subsequence does not reorder elements, and that it need not contain consecutive elements from the larger sequence. Figure 9 conveys this information by presenting a recursive definition for $\preceq$. Section 9 provides more information about how to read such auxiliary specifications.

```
Subsequence(E): trait
  includes Sequence(E)
  introduces __≼__: Seq[E], Seq[E] → Bool
  asserts with e, e1, e2: E, s, s1, s2: Seq[E]
    {} ≼ s;
    ¬((s ⊢ e) ≼ {});
    (s1 ⊢ e1) ≼ (s2 ⊢ e2) ⇔ (s1 ⊢ e1) ≼ s2 ∨ (s1 ≼ s2 ∧ e1 = e2)
```

Figure 9: Auxiliary specification with recursive definition of subsequence operator

An abbreviated form of nondeterministic choice can be used in the assignment statement to express the fact that a transition can change the value of a state variable, without specifying what the new value may be. Such a nondeterministic choice consists of the keyword **choose** alone, without a subsequent variable or **where** clause. The statement `x :=` **choose** is equivalent to the somewhat longer statement `x :=` **choose** `y` **where** `true`. Both of these statements give a transition a license to change the value of the state variable `x`. As described below, constraints on the new values for modified variables, if any, can be given in a **so that** clause for the entire effect.

Conditional statements A conditional statement is used to select which of several program segments to execute in a larger program. It starts with the keyword **if** followed by a predicate, the keyword **then**, and a program segment; it ends with the keyword**fi**. In between, there can be any number of **elseif** clauses (each of which contains a predicate, the keyword **then**, and a program segment), and there can be a final **else** clause (which also contains a program segment). Figure 10 illustrates the use of a conditional statement in defining an automaton that distributes input values into one of three sets. Section 8.2 describes the set data type and the operators `{}` and `insert`.

For ***statements*** A **for** statement is used to perform a program segment once for each value of a variable that satisfies a given condition. It starts with the keyword **for** followed by a variable, a clause describing a set of values for this variable, the keyword **do**, a program segment, and the keyword **od**.

Figure 11 illustrates the use of a **for** statement in a high-level description of a multicast algorithm. Its first line defines the `Packet` data type to consist of triples `[contents, source, dest]`, in which `contents` represents a message, `source` the `Node` from which the message originated, and `dest` the set of `Nodes` to which the message should be delivered. The state of the multicast algorithm consists of a multiset `network`, which represents the packets currently in transit, and an array `queue`, which represents, for each `Node`, the sequence of packets delivered to that `Node`, but not yet read by the `Node`.

The `mcast` action inserts a new packet in the `network`; the notation `[m, i, I]` is defined by the tuple data type (Section 9.8) and the `insert` operator by the multiset data type (Section 8.3). The `deliver` action, which is described using a **for** statement, distributes a packet to all nodes in its destination set (by appending the packet to the queue for each node in the destination set and

```
automaton Distribute
  signature
    input get(i: Int)
  states
    small:  Set[Int] := {},
    medium: Set[Int] := {},
    large:  Set[Int] := {},
    bound1: Int,
    bound2: Int
    so that bound1 < bound2
  transitions
    input get(i)
      eff if i  < bound1 then small := insert(i, small)
          elseif i < bound2 then medium := insert(i, medium)
          else large := insert(i, large)
          fi
```

Figure 10: Example of a conditional statement

```
type Packet = tuple of contents: Message, source: Node, dest: Set[Node]

automaton Multicast
  signature
    input    mcast(m: Message, i: Node, I: Set[Node])
    internal deliver(p: Packet)
    output   read(m: Message, j: Node)
  states
    network: Mset[Packet] := {},
    queue:   Array[Node, Seq[Packet]]
    so that ∀ i: Node (queue[i] = {})
  transitions
    input mcast(m, i, I)
      eff network := insert([m, i, I], network)
    internal deliver(p)
      pre p ∈ network
      eff for j: Node in p.dest do queue[j] := queue[j] ⊢ p od;
          network := delete(p, network)
    output read(m, j)
      pre queue[j] ≠ {} ∧ head(queue[j]).contents = m
      eff queue[j] := tail(queue[j])
```

Figure 11: Example showing use of a **for** statement

then deleting the packet from the `network`). The `read` action receives the contents of a packet at a particular `Node` by removing that packet from the `queue` of delivered packets at that `Node`.

In general, the clause describing the set of values for the control variable in a **for** statement consists either of the keyword **in** followed by an expression denoting a set (Section 8.2) or multiset (Section 8.3) of values of the appropriate type, or of the keywords **so that** followed by a predicate. The program following the keyword **do** is executed once for each value in the set or multiset following the keyword **in**, or once for each value satisfying the predicate following the keywords **so that**. These versions of the program are executed in an arbitrary order. However, IOA restricts the form of the program so that the effect of the **for** statement is independent of the order in which the versions of the program are executed.

Using predicates on states to specify effects The results of a program can be constrained by a predicate relating the values of state variables after a transition has occurred to the values of state variables before the transition began. Such a predicate is particularly useful when the program contains the nondeterministic **choose** operator. For example,

```
input crash
  eff buffer := choose
      so that buffer′ ⪯ buffer
```

is an alternative, but equivalent way of describing the `crash` action in `LossyChannel` (Figure 8). The assignment statement indicates that the `crash` action can change the value of the state variable `buffer`. The predicate in the **so that** clause constrains the new value of `buffer` in terms of its old value. A primed state variable in this predicate (i.e., `buffer′`) indicates the value of the variable in the post-state; an unprimed state variable (i.e., `buffer`) indicates its value in the pre-state. For another example,

```
 eff name[i] := choose;
     name[j] := choose
     so that name′[i] = name[j] ∧ name′[j] = name[i]
```

is an alternative way of writing the effect clause of the `swap` action in `Shuffle` (Figure 7). The assignment statements indicate that the array `name` may be modified at indices `i` and `j`, and the **so that** clause constrains the modifications. This notation allows us to eliminate the `temp` state variable needed previously for swapping.

There are important differences between **where** and **so that** clauses. A **where** clause can be attached to a nondeterministic **choose** operator in a single assignment statement to restrict the value chosen by that operator; variables appearing in a **where** clause denote values in the state before the assignment statement is executed. A **so that** clause can be attached to an entire **eff** clause; unprimed variables appearing in a **so that** clause denote values in the state before the transition represented by the entire **eff** clause occurs, and primed variables denote values in the state after the transition has occurred.

4.4.4 Choose parameters

Two kinds of parameters can be specified for a transition: ordinary parameters, corresponding to those in the automaton's signature, and additional "**choose** parameters," which provide a convenient way to relate the postcondition for a transition to its precondition. Figure 12 illustrates the use of **choose** parameters.

The automaton `LossyBuffer` represents a message channel that loses a message each time it transmits one. The state of the automaton consists of a multiset `buff` of messages of type `M`. The input action for the channel, `get(m)`, simply adds the message `m` to `buff`. The output action, `put(m)`, delivers `m` while dropping another message, given by the **choose** parameter `n`. The precondition

```
automaton LossyBuffer(M: type)
  signature
    input  get(m: M)
    output put(m: M)
  states
    buff: Mset[M] := {}
  transitions
    input get(m)
      eff buff := insert(m, buff)
    output put(m)
      choose n: M
      pre m ∈ buff ∧ n ∈ buff ∧ (m ≠ n ∨ count(n, buff) > 1)
      eff buff := delete(m, delete(n, buff))
```

Figure 12: Example of the use of **choose** parameters

ensures that both `m` and `n` a remembers of the multiset `buff` and, if `m` and `n` happen to be the same message, that `buff` contains two copies of this message.

Choose parameters provide syntactic sugar for defining transitions. It is possible to define transitions without them by using explicit quantification. For example, the transition for the `put` action in Figure 12 can be rewritten as follows:

```
  output put(m)
    pre ∃ n: M (m ∈ buff ∧ n ∈ buff ∧ (m ≠ n ∨ count(m, buff) > 1))
    eff buff := choose
        so that ∃ n: M (m ∈ buff ∧ n ∈ buff ∧ (m ≠ n ∨ count(m, buff) > 1)
                        ∧ buff′= delete(m, delete(n, buff)))
```

In general, to eliminate **choose** parameters, one quantifies them explicitly in the precondition for the transition, and then repeats the quantified precondition as part of the effect.

4.5 Tasks

A final, but optional part in the description of an I/O automaton is a partition of the automaton's output and internal actions into a set of disjoint tasks. This partition is indicated by the keyword **tasks** followed by a list of the sets in the partition. If the keyword **tasks** is omitted, and no task partition is given, all output and internal actions are presumed to belong to the same task.

To see why tasks are useful, consider the automaton `Shuffle` described in Figure 7. The traces of this automaton can be either infinite sequences of `swap` actions, a finite sequence of `swap` actions, or a finite sequence of `swap` actions followed by a single `deal` action: nothing in the description in Figure 7 requires that a `deal` action ever occur. By adding

```
  tasks
    {swap(i, j) for i: cardIndex, j: cardIndex};
    {deal(a) for a: Array[cardIndex, String]}
```

to the description of `Shuffle`, we can place all `swap` actions in one task (or thread of control) and all `deal` actions in another. The definition of a *fair* execution of an I/O automaton requires that, whenever a task remains enabled, some action in that task will eventually be performed. Thus this task partition for `Shuffle` prevents `swap` actions from starving a `deal` action in any fair execution. There are no fairness requirements, however, on the actions within the same task: the description of `Shuffle` does not require that every pair of elements in the array will eventually be interchanged.

Variables appearing in task definitions must be introduced using the keyword **for**, either within the braces defining individual tasks (as illustrated for `Shuffle`) or outside the braces. For example the task partition

```
 tasks {deliver(p) for p: Packet}; {read(m, j) for m: Message} for j: Node
```

for the `Multicast` automaton places the `read` actions for different nodes in different tasks, so that the execution of `read` actions for one node cannot starve execution of `receive` actions for another. The values of variables appearing in task definitions can be constrained further by **where** clauses following the **for** clauses.

*Editorial note: Do we want to allow more general set-theoretic notations for defining tasks??? For example, do we want to allow {foo(i) **for** i: I} ∪ {bar(i) **for** i: I} in addition to or in place of {foo(i), bar(i) **for** i: I}?*

5 IOA notations for operations on automata

We often wish to describe new automata in terms of previously defined automata. IOA provides notations for composing several automata, for hiding some output actions in an automaton, and for specializing parameterized automata.[6]

5.1 Composition

We illustrate composition by describing the LeLann-Chang-Roberts (LCR) leader election algorithm as a composition of process and channel automata.

In this algorithm, a finite set of processes arranged in a ring elect a leader by communicating asynchronously. The algorithm works as follows. Each process sends a unique string representing its name, which need not have any special relation to its index, to its right neighbor. When a process receives a name, it compares it to its own. If the received name is greater than its own in lexicographic order, the process transmits the received name to the right; otherwise the process discards it. If a process receives its own name, that name must have traveled all the way around the ring, and the process can declare itself the leader.

Figure 13 describes such a process, which is parameterized by the type `I` of process indices and by a process index `i`. The **assumes** clause identifies an auxiliary specification, `RingIndex` (Figure 14), that imposes restrictions on the type `I`. This specification requires that there be a ring structure on `I` induced by the operators `first`, `right`, and `left`, and that `name` provide a one-one mapping from indices of type `I` to names of type `String`.

The **type** declaration on the first line of Figure 13 declares `Status` to be an enumeration (Section 9.8) of the values `waiting`, `elected`, and `announced`.

The automaton `Process` has two state variables: `pending` is a multiset of strings, and `status` has type `Status`. Initially, `pending` is set to `{name(i)}` and `status` to `waiting`. The input action `receive(m, left(i), i)` compares the name received from the `Process` automaton to the left of this automaton in the ring and the name of the automaton itself. There are two output actions: `send(m, i, right(i))`, which simply sends a message in `pending` to the `Process` automaton on the right in the ring, and `leader(m, i)`, which announces successful election. The two kinds of output actions are placed in separate tasks, so that a `Process` automaton whose status is `elected` must eventually perform a `leader` action.

Editorial note: Should we say something about why the transitions are specified as send(m, i, j) and receive(m, j, i)? The signature of the automaton restricts the values of j to be left(i) and checking to ensure that this convention is being respected?

[6]Eventually IOA will also provide notations for renaming actions.