IOA User Guide and Reference Manual[1]

Stephen J. Garland, Nancy A. Lynch, Joshua A. Tauber, and Mandana Vaziri
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

August 14, 2003

[1] *Editorial note: Additions are currently being made to this document. These additions largely concern the formal semantics of IOA, usage of software tools with IOA, and annotations of IOA programs for use with these tools. Notes within the document indicate sections currently being extended.*

Abstract

IOA is a simple formal language for modeling distributed systems as collections of interacting state machines, called input/output automata. The IOA Toolkit supports a range of validation methods, including simulation and machine-checked proofs. This manual and reference guide defines the IOA language.

Research on IOA has been supported in part by the following sponsors:

- DARPA under contracts N00014-92-J-1795 and N66001-99-2-891702 monitored by the Office of Naval Research and contract F19628-95-C-0018 monitored by Hanscom Air Force Base;
- the National Science Foundation under grants CCR-9504248 and CCR-9225124;
- by the Air Force Office of Scientific Research and the Office of Naval Research under contract F49620-94-1-0199;
- NTT under contract MIT9904-12; and
- Draper Laboratory under IRaD project 5005.

Contents

Part I
IOA Tutorial

The Input/Output (I/O) automaton model, developed by Nancy Lynch and Mark Tuttle [11], models components in asynchronous concurrent systems as labeled transition systems. Lynch's book, *Distributed Algorithms* [10], describes many algorithms in terms of I/O automata and contains proofs of various properties of these algorithms.

IOA is a precise language for describing I/O automata and for stating their properties. It extends and formalizes the descriptive notations used in *Distributed Algorithms*, uses Larch specifications [8] to define the semantics of abstract data types and I/O automata, and supports a variety of analytic tools. These tools range from light weight tools, which check the syntax of automaton descriptions, to medium weight tools, which simulate the action of an automaton, and to heavier weight tools, which provide support for proving properties of automata.

The document is organized as follows. Part I contains an informal introduction to I/O automata and a tutorial for IOA. The tutorial consists largely of examples that illustrate different aspects of the language; reading it should be sufficient to begin writing complete IOA descriptions. Part II describes the data types available for use in IOA descriptions. Finally, Parts III and IV present the formal syntax and semantics of the language.

1 Introduction

I/O automata provide a mathematical model suitable for describing asynchronous concurrent systems. The model provides a precise way of describing and reasoning about system components that interact with each other and that operate at different speeds. It also permits components that have been described as I/O automata to be composed into larger automata.

1.1 I/O automata

An I/O automaton is a simple type of state machine in which the transitions are associated with named *actions*. The actions are classified as either *input*, *output*, or *internal*. The inputs and outputs are used for communication with the automaton's environment, whereas internal actions are visible only to the automaton itself. The input actions are assumed not to be under the automaton's control, whereas the automaton itself controls which output and internal actions should be performed.

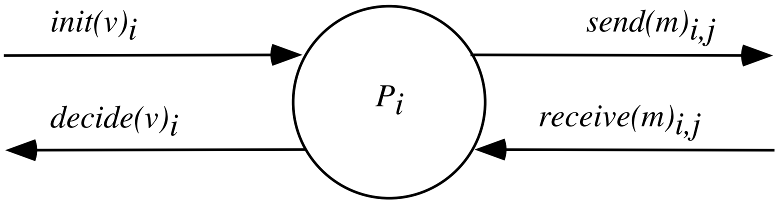

Figure 1.1: A process

A typical example[1] of an I/O automaton is a process in an asynchronous distributed system. Figure 1.1 shows the interface of one such process. The circle represents the automaton, named P_i,

[1]This example is essentially the same as the example in *Distributed Algorithms* [10], Chapter 8.

where i is a process index, and the arrows represent input and output actions. An incoming arrow is an input action, and an outgoing arrow is an output action. Internal actions are not shown. Process P_i can receive inputs of the form $init(v)_i$, each of which represents the receipt of an input value v, and it can produce outputs of the form $decide(v)_i$, each of which represents a decision on the value of v. In order to reach a decision, process P_i may communicate with other processes using a message passing system. P_i's interface to the message system consists of output actions of the form $send(m)_{i,j}$, each of which represents sending a message m to some process named P_j, and input actions of the form $receive(m)_{i,j}$, each of which represents receiving a message m from process P_j. When P_i performs any of the indicated actions (or any internal action), it may also change state.

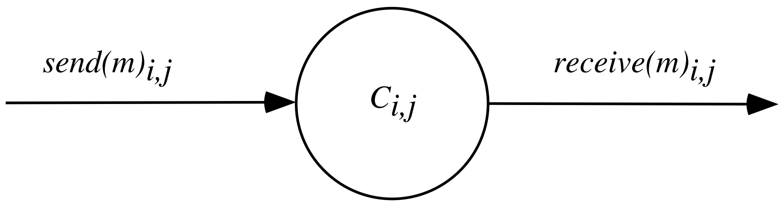

Figure 1.2: A channel

Another example of an I/O automaton is a FIFO message channel. Figure 1.2 shows the interface of a typical channel automaton, $C_{i,j}$, where i and j are process indices. Its input actions have the form $send(m)_{i,j}$, and its output actions have the form $receive(m)_{i,j}$.

Process and channel automata can be composed as shown in Figure 1.3, by matching the output actions of one automaton with the input actions of another. Thus, each output action $send(m)_{i,j}$ of a process automaton is matched and performed together with an input action $send(m)_{i,j}$ of some channel automaton, and each input action $receive(m)_{i,j}$ of a process automaton is matched and performed together with an output action $receive(m)_{i,j}$ of some other channel automaton. Actions are performed one at a time, indivisibly, in any order.

More precisely, an I/O automaton A consists of the following five components:

- a *signature*, which lists the disjoint sets of input, output, and internal actions of A,
- a (not necessarily finite) set of *states*, usually described by a collection of state variables,
- a set of *start* (or *initial*) *states*, which is a non-empty subset of the set of all states,
- a *state-transition relation*, which contains triples (known as *steps* or *transitions*) of the form (state, action, state), and
- an optional set of *tasks*, which partition the internal and output actions of A.

An action π is said to be *enabled* in a state s if there is another state s' such that (s, π, s') is a transition of the automaton. Input actions are enabled in every state; i.e., automata are not able to "block" input actions from occurring. The *external* actions of an automaton consist of its input and output actions.

The transition relation is usually described in *precondition-effect* style, which groups together all transitions that involve a particular type of action into a single piece of code. The precondition is a predicate on the state indicating the conditions under which the action is permitted to occur. The effect describes the changes that occur as a result of the action, either in the form of a simple program or in the form of a predicate relating the pre-state and the post-state (i.e., the states before and after the action occurs). Actions are executed indivisibly.

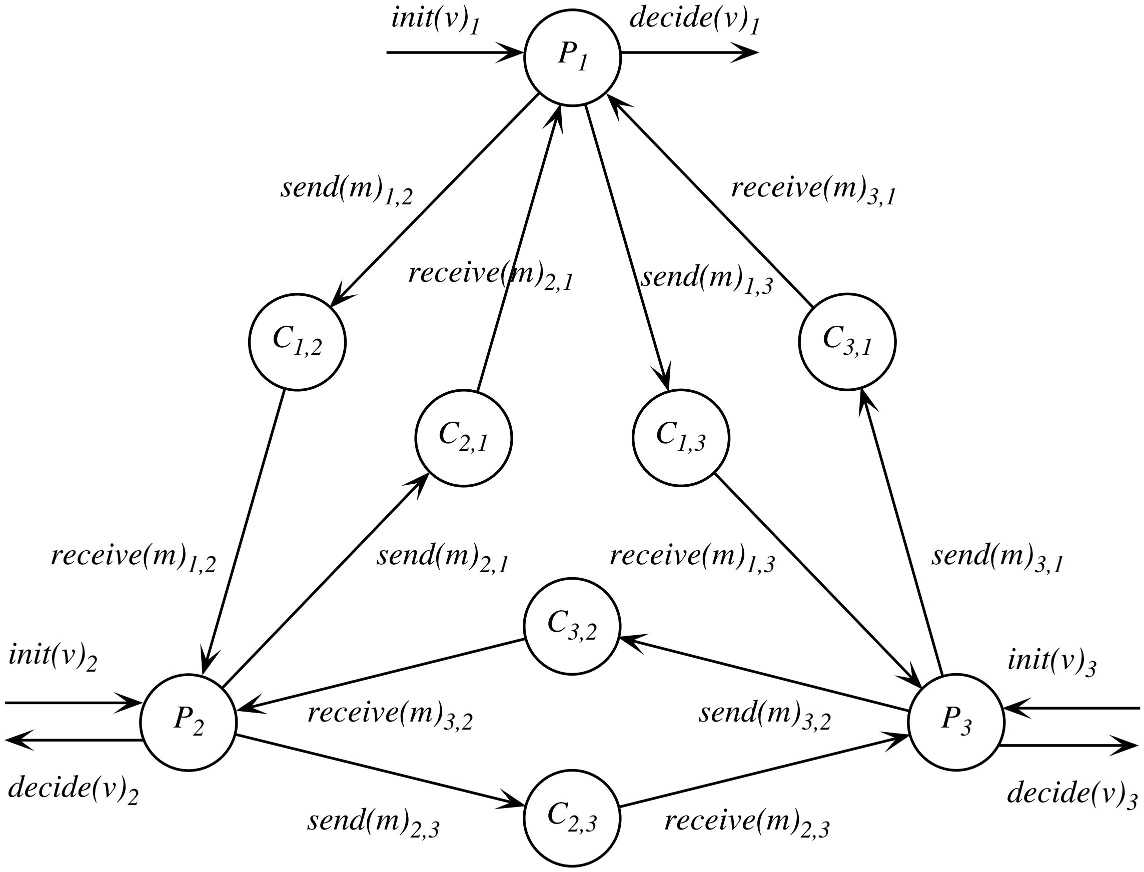

Figure 1.3: Composing channel and process automata

1.2 Executions and traces

An *execution fragment* of an I/O automaton is either a finite sequence s_0, π_1, s_1, π_2, ..., π_n, s_n, or an infinite sequence s_0, π_1, s_1, π_2, ..., of alternating states s_i and actions π_i such that $(s_i, \pi_{i+1}, s_{i+1})$ is a transition of the automaton for every $i \geq 0$. An *execution* is an execution fragment that begins with a start state. A state is *reachable* if it occurs in some execution. The *trace* of an execution is the sequence of external actions in that execution.

The task partition is an abstract description of "tasks" or "threads of control." It is used to define *fairness conditions* on an execution of the automaton; these conditions require the automaton to continue, during its execution, to give fair turns to each of its tasks. A task is said to be *enabled* in a state if some action in the task is enabled in that state. In a *fair execution*, whenever some task remains enabled, some action in that task will eventually be performed. Thus, in fair executions, actions in one task partition do not prevent actions in another from occurring. If no task partition is specified, then all actions are assumed to belong to a single task.

1.3 Operations on automata

The operation of *composition* allows an automaton representing a complex system to be constructed by composing automata representing individual system components. The composition identifies actions with the same name in different component automata. When any component automaton performs a step involving an action π, so do all component automata that have π in their signatures. The *hiding* operation "hides" output actions of an automaton by reclassifying them as internal actions; this prevents them from being used for further communication and means that they are no longer included in traces. The *renaming* operation changes the names of an automaton's actions, to facilitate composing that automaton with others that were defined with different naming conventions.

1.4 Properties of automata

An *invariant* of an automaton is any property that is true in all reachable states of the automaton.

An automaton A is said to *implement* an automaton B provided that A and B have the same input and output actions and that every trace of A is a also trace of B. In order to show that A implements B, one can use a *simulation relation* between states of A and states of B such that, loosely speaking, every start state of A is related to a start state of B and every reachable state of A is related to a state of B reached by the same series of external actions.

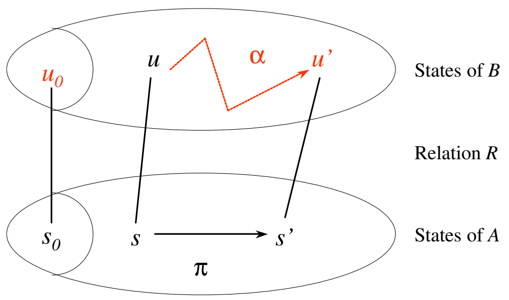

Figure 1.4: Forward simulation relation

For the purpose of a formal definition, we assume that A and B have the same input and output actions. A relation R between the states of A and B is a *forward simulation*[2] with respect to invariants I_A and I_B of A and B if

- every start state of A is related (via R) to a start state of B, and
- for all states s of A and u of B satisfying the invariants I_A and I_B such that $R(s, u)$, and for every step (s, π, s') of A, there is an execution fragment α of B starting with u, containing the same external actions as π, and ending with a state u' such that $R(s', u')$.

A general theorem is that A implements B if there is a forward simulation from A to B.

Similarly, a relation R between the states of A and B is a *backward simulation*[3] with respect to invariants I_A and I_B of A and B if

- every state of A that satisfies I_A corresponds (via R) to some state of B that satisfies I_B,
- if a start state s of A is related (via R) to a state u of B that satisfies I_B, then u is a start state of B, and
- for all states s, s' of A and u' of B satisfying the invariants such that $R(s', u')$, and for every step (s, π, s') of A, there is an execution fragment α of B ending with u', containing the same external actions as π, and starting with a state u satisfying I_B such that $R(s, u)$.

Another general theorem is that A implements B if there is an *image-finite* backward simulation from A to B. Here, a relation R is image-finite provided that for any x there are only finitely many y such that $R(x, y)$. Moreover, the existence of any backward simulation from A to B implies that all finite traces of A are also traces of B.

2 Using IOA to formalize descriptions of I/O automata

We illustrate the nature of I/O automata, as well as the use of the language IOA to define the automata, by a few simple examples. Figure 2.1 contains a simple IOA description for an automaton, `Adder`, that gets two integers as input and subsequently outputs their sum. The first line declares the name of the automaton. The remaining lines define its operation. The signature consists of input actions `add(i, j)`, one for each pair of values of `i` and `j`, and output actions `result(k)`, one for each value of `k`. The type `Int`, used to represent integers, is a built-in type in IOA (see Section 7.2).

The automaton `Adder` has two state variables: `value` is an integer that is used to hold a sum, and `ready` is a boolean that is set to `true` whenever a new sum has been computed. The initial value of `value` is arbitrary since it is not specified; `ready` is initially `false`.

The transitions of the automaton `Adder` are given in precondition/effect style. The input action `add(i, j)` has no precondition, which is equivalent to its having `true` as a precondition. This is the case for all input actions; that is, every input action in every automaton is enabled in every state. The effect of `add(i, j)` is to change `value` to the sum of `i` and `j` and to set `ready` to `true`. The output action `result(k)` can occur only when it is enabled, that is, only in states where its precondition `k` $=$ `value` $\wedge$ `ready` is true. Its effect is to set `ready` back to `false`. Traces of `Adder` are sequences of external actions such as

[2]In some previous work such relations are called *weak* forward simulations.

[3]In some previous work such relations are called *weak* backward simulations.

```
automaton Adder
  signature
    input  add(i, j: Int)
    output result(k: Int)
  states
    value: Int,
    ready: Bool := false
  transitions
    input add(i, j)
      eff value := i + j;
          ready  := true
    output result(k)
      pre k = value ∧ ready
      eff ready := false
```

Figure 2.1: IOA description of an adder

`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 ReliableChannel(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 2.2: IOA description of a reliable communication channel

Another simple automaton, `ReliableChannel`, is shown in Figure 2.2. This automaton represents a reliable communication channel, as illustrated in Figure 1.2, 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 `i` and `j` are terms (not variables, as in `adder`), whose values are fixed by the values of the automaton's parameters.

The state of the automaton `ReliableChannel` 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 `{}`, $\vdash$, `head`, and `tail`.

The input action `send(m, i, j)` has the effect of appending `m` to `buffer` (here, $\vdash$ 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[I1, ... , In, E]` is an n-dimensional array of elements of type `E` indexed by elements of types `I1, ... , In`.
- `Map[D1, ... , Dn, R]` is a finite partial mapping of elements of an n-dimensional domain with type `D1` $\times \cdots \times$ `Dn` to elements of a range with type `R`. Mappings differ from arrays in that they are defined only for finitely many elements of their domains (and hence 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`.
- `Null[E]` is isomorphic to `E` extended by a single element `nil`.

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. Table 3.1 shows 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.

<table>
<tr><th colspan="3">Logical Operator</th><th colspan="3">Datatype Operator</th></tr>
<tr><th>Symbol</th><th>Meaning</th><th>Input</th><th>Symbol</th><th>Meaning</th><th>Input</th></tr>
<tr><td>$\forall$</td><td>For all</td><td><code>\A</code></td><td>$\leq$</td><td>Less than or equal</td><td><code><=</code></td></tr>
<tr><td>$\exists$</td><td>There exists</td><td><code>\E</code></td><td>$\geq$</td><td>Greater than or equal</td><td><code>>=</code></td></tr>
<tr><td>$\neg$</td><td>Not</td><td><code>~</code></td><td>$\in$</td><td>Member of</td><td><code>\in</code></td></tr>
<tr><td>$\neq$</td><td>Not equals</td><td><code>~=</code></td><td>$\notin$</td><td>Not a member of</td><td><code>\notin</code></td></tr>
<tr><td>$\wedge$</td><td>And</td><td><code>/\</code></td><td>$\subset$</td><td>Proper subset of</td><td><code>\subset</code></td></tr>
<tr><td>$\vee$</td><td>Or</td><td><code>\/</code></td><td>$\subseteq$</td><td>Subset of</td><td><code>\subseteq</code></td></tr>
<tr><td>$\Rightarrow$</td><td>Implies</td><td><code>=></code></td><td>$\supset$</td><td>Proper superset of</td><td><code>\supset</code></td></tr>
<tr><td>$\Leftrightarrow$</td><td>If and only if</td><td><code><=></code></td><td>$\supseteq$</td><td>Superset of</td><td><code>\supseteq</code></td></tr>
<tr><td></td><td></td><td></td><td>$\vdash$</td><td>Append element</td><td><code>|-</code></td></tr>
<tr><td></td><td></td><td></td><td>$\dashv$</td><td>Prepend element</td><td><code>-|</code></td></tr>
</table>

Table 3.1: Typographical conventions

4 IOA descriptions for primitive automata

Explicit descriptions of primitive automata specify 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 2.1 and 2.2). As illustrated in Figure 2.2, the name may be followed by a list of formal parameters enclosed within parentheses. There are two kinds of automaton parameters. An individual parameter (such as `i:Index` and `j:Index`) consists of an identifier with its associated type, and it denotes a fixed element of that type. A type parameter (such as `M:`**`type`** and `Channel:`**`type`**) consists of an identifier followed by the keyword **type**, and it denotes 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. There are two kinds of action parameters. A varying parameter (such as `m:M` in Figure 2.2) consists of an identifier with its associated type, and it denotes an arbitrary element of that type. A fixed parameter (such as **const** `i` and **const** `j`) consists of the keyword **const** followed by a term denoting a fixed element of its type. Neither kind of parameter can have **type** as its type. Each entry in the signature denotes a set of actions, one for each assignment of values to its varying parameters.

[4]In the future, IOA may also allow relations (e.g., $<$) and functions as automaton parameters.

It is possible to constrain the values of the varying 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 either 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 their initial values cannot refer to the value of any state variable.

A nondeterministic choice (of the form **choose** *variable* **where** *predicate*) selects an arbitrary value of the variable that satisfies the predicate. When a nondeterministic choice is used to initialize a state variable, there must be some value of the variable that satisfies the predicate. If the predicate is true for all values of the 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
  transitions
    output result(i)
      pre i = num
```

Figure 4.1: Example of nondeterministic choice of initial value for state variable

For example, in the automaton `Choice` (Figure 4.1), the state variable `num` is initialized nondeterministically to some value of the variable `n` that satisfies the predicate `1 ≤ n ∧ n ≤ 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). Traces of the automaton `Choice` are sequences of identical output actions (in which the parameter `i` has the value assigned initially to the state variable `num`).

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 keyword **initially** followed by a predicate (involving state variables and automaton parameters), as illustrated by the definition of the automaton `Shuffle` in Figure 4.2.[5]

In Figure 4.2, 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`. The constraint following **initially** limits only the initial values of the state variables, not their subsequent values. (Note

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

```
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
    initially ∀ i: cardIndex (i ≠ 52 ∧ i ≠ cut ⇒ name[i] < name[succ(i)])
              ∧ name[52] < name[1]
  transitions
    internal swap(i, j; local temp: String)
      pre ¬dealt
      eff temp := name[i];
          name[i] := name[j];
          name[j] := temp
    output deal(a)
      pre ¬dealt ∧ a = name
      eff dealt := true
```

Figure 4.2: Example of a constraint on initial values for state variables

that the scope of the **initially** clause is the entire set of state variable declarations.) The type `Array[cardIndex, String]` of the state variable `name` consists of 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.

When the type of a state variable is an `Array`, `Map`, or **tuple** (Section 9.8), IOA also treats the elements of the array or mapping, 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 mapping or of the fields in the tuple.

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, an optional **where** clause, an optional precondition, and an optional effect.

4.4.1 Transition parameters

Two kinds of parameters can be specified for a transition: ordinary parameters, corresponding to those in the automaton's signature, and additional **local** parameters (described below in Section 4.4.4). The ordinary parameters accompanying an action name in a transition definition must match those accompanying the action name in the automaton's signature, both in number and in type. However, the keyword **const** does not appear in transition parameters, and all transition parameters are treated as terms.

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 2.2,

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

in the signature of `ReliableChannel` is shortened to **input** `send(m, i, j)` in the transition definition. (See Section 16.3 for more details.)

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 4.2) can be split into two parts:

```
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 **local** 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 there are some values for the **local** parameters of one of its transition definitions that satisfy both the **where** clause and, together with the state variables, the precondition for that transition definition. 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 or **local** variable. The statement consists of the state or **local** 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 or mapping used as a state or **local** variable, or the fields in a tuple used as a state or **local** variable, are themselves considered as separate 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 or **local** variable. The value of the expression is defined mathematically, rather than computationally, in the state before the assignment statement is executed.[6] The value of the expression then becomes the value of the state or **local** 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 or **local** variable other than the one 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 4.3: IOA description of a lossy communication channel

The definition of the `crash` action in the `LossyChannel` automaton (Figure 4.3) illustrates the use of the **choose** ... **where** construct to constrain the sequence chosen as 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 2.2) 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 4.3 identifies an auxiliary specification (Figure 4.4), 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 introduce it in an auxiliarly specification that defines both its syntax (i.e., that it is a binary relation on sequences) and its properties (i.e., that a subsequence does not reorder elements, and that it need not contain consecutive elements from

[6]If a program consists of more than a single assignment statement, then the states before and after the assignment statements in the program may be intermediate states, which do not appear in the execution fragments of the automaton.

the larger sequence). Figure 4.4 conveys this information by presenting a recursive definition for $\preceq$. Section 9 provides more information about how to read and write 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 4.4: 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` arbitrarily. As described below, constraints on the new values for modified variables, if any, can be given in an **ensuring** 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 4.5 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`.

```
automaton Distribute
  signature
    input get(i: Int)
  states
    small:  Set[Int] := {},
    medium: Set[Int] := {},
    large:  Set[Int] := {},
    bound1: Int,
    bound2: Int
    initially 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 4.5: Example of a conditional statement

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**.

```
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]]
    initially ∀ 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 4.6: Example showing use of a **for** statement

Figure 4.6 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 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`.

There are two ways to describe the set of values for the control variable in a **for** statement. The first consists 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, in which case the program following the keyword **do** is executed once for each value in the set or multiset. The second consists of the keyword **where** followed by a predicate, in which case the program is executed once for each value satisfying the predicate. These executions of the program occur in an arbitrary order. However, IOA requires that the effect of a **for** statement be independent of the order in which executions of its program occur.

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
      ensuring buffer′ ⪯ buffer
```

is an alternative, but equivalent way of describing the `crash` action in `LossyChannel` (Figure 4.3). The assignment statement indicates that the `crash` action can change the value of the state variable `buffer`. The predicate in the **ensuring** 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
    ensuring name′[i] = name[j] ∧ name′[j] = name[i]
```

is an alternative way of writing the effect clause of the `swap` action in `Shuffle` (Figure 4.2). The assignment statements indicate that the array `name` may be modified at indices `i` and `j`, and the **ensuring** clause constrains the modifications. This notation allows us to eliminate the **local** variable `temp` needed previously for swapping.

There are important differences between **where** clauses attached to nondeterministic **choose** operators and **ensuring** clauses. A **where** clause restricts the value chosen by a **choose** operator in a single assignment statement, and variables appearing in the **where** clause denote values in the state before the assignment statement is executed. An **ensuring** clause can be attached to an entire **eff** clause; unprimed variables appearing in an **ensuring** 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 Local parameters

In addition to its ordinary parameters, a transition definition can contain "**local** parameters," which can be used for two purposes. As illustrated in Figure 4.2, they can be used as temporary state variables. In addition, they can be used to relate the postcondition for a transition to its precondition, as illustrated in Figure 4.7.

```
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; local n: M)
      pre m ∈ buff ∧ n ∈ buff ∧ (m ≠ n ∨ count(n, buff) > 1)
      eff buff := delete(m, delete(n, buff))
```

Figure 4.7: Example of the use of **local** 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 **local** parameter `n`. The precondition 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.

Local parameters to which no values are assigned 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 4.7 can be rewritten as follows:

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

In general, to eliminate **local** parameters to which no values are assigned, 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 (see Section 1.1. 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 4.2. 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 4.2 requires that a `deal` action ever occur. By adding

```
tasks {swap(i, j)}; {deal(a)}
```

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. If we wished to impose this stronger requirement, we could place each `swap` action into a separate task by amending the task definition to

```
tasks {swap(i, j)} for i, j: Int; {deal(a)}
```

The keyword **for** indicates that there is one task for each assignment of values to the variables following the keyword. For another example, the task partition

```
tasks {deliver(p)}; {read(m, j)} 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, either within the braces or following the **for** clauses. Variables that appear in a task definition, but not in the **for** clause for that task, are universally quantified: they take on all values allowed by the **where** clauses in the action signatures and in the task definition.

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.[7]

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.

```
type Status = enumeration of waiting, elected, announced

automaton Process(I: type, i: I)
  assumes RingIndex(I, String)
  signature
    input  receive(m: String, const left(i), const i)
    output send(m: String, const i, const right(i)) where m ≥ name(i),
           leader(const name(i), const i)
  states
    pending: Mset[String] := {name(i)},
    status:  Status := waiting
  transitions
    input receive(m, j, i) where m > name(i)
      eff pending := insert(m, pending)
    input receive(m, j, i) where m < name(i)
    input receive(name(i), j, i)
      eff status := elected
    output send(m, i, j)
      pre m ∈ pending
      eff pending := delete(m, pending)
    output leader(m, i)
      pre status = elected
      eff status := announced
  tasks
    {send(m, j, right(j))};
    {leader(m, j)}
```

Figure 5.1: IOA specification of election process

Figure 5.1 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 5.2), that imposes restrictions on the type `I`. This specification requires that there be

[7] In the future IOA may also provide notations for renaming actions.

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`.

```
RingIndex(I, J): trait
  introduces
     first:         → I
     left, right: I → I
     name:          I → J
  asserts with i, j: I
    sort I generated by first, right;
    ∃ i (right(i) = first);
    right(i) = right(j) ⇔ i = j;
    left(right(i)) = i;
    name(i) = name(j) ⇔ i = j
  implies with i: I
    right(left(i)) = i
```

Figure 5.2: Auxiliary specification for a finite ring of process identifiers

The **type** declaration on the first line of Figure 5.1 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`. As described above, there are three possible transitions for the input action `receive(m,` **const** `left(i),` **const** `i)`, depending on how the string `m` received from the `Process` automaton to the left of automaton `i` compares with the name of automaton `i`. These transitions are described in three separate transition definitions; they could just as well have been described in a single definition using a conditional statement. The value of the first parameter of `receive` is constrained by **where** clauses in the first two transition definitions and is fixed in the third. The parameter `j` in each of these transition definitions is constrained to equal `left(i)` by the the action signature. Also as described above, there are two kinds of output actions: `send(m, i, right(i))`, which simply sends a message in `pending` to the `Process` automaton to the right in the ring, and `leader(m, i)`, which announces successful election. The value of the parameter `m` in `leader` action is constrained to equal `name(i)` by the signature of that action. Finally, 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.

```
automaton LCR(I: type)
  assumes RingIndex(I, String)
  components
    P[i: I]: Process(I, i);
    C[i: I]: ReliableChannel(String, I, i, right(i))
```

Figure 5.3: IOA specification of LCR algorithm

The full LCR leader election algorithm is described in Figure 5.3 as a composition of a set of process automata connected in a ring by reliable communication channels (Figures 1.2 and 2.2). The **assumes** statement on the first line repeats the assumption about the type `I` of process indices in Figure 5.1. The keyword **components** introduces a list of named components: one `Process` automaton, `P[i]`, and one `ReliableChannel` automaton, `C[i]`, for each element `i` of type `I`. The component `C[i]` is obtained by instantiating the type parameters `M` and `Index` for the

`ReliableChannel` automaton (Figure 2.2) with the actual types `String` and `I` of messages and process indices, and the parameters `i` and `j` with the values `i` and `right(i)`, so that channel `C[i]` connects process `P[i]` to its right neighbor. The output actions `send(m, i, right(i))` of `P[i]` are identified with the input actions `send(m, i, right(i))` of `C[i]`, and the input actions `receive(m, left(i), i)` of `P[i]` are identified with the output actions `receive(m, left(i), i)` of `C[left(i)]`, which is `ReliableChannel(String, I, left(i), i)` because `RingIndex` implies that `right(left(i)) = i`. Since all input actions of the channel and process subautomata are identified with output actions of other subautomata, the composite automaton contains only output actions.

5.2 Specialization

A parameterized automaton description defines a set of automata rather than a single automaton. For example, `LCR` defines a set of automata, operating on rings of varying size, rather than a single automaton, operating on a ring with a fixed size. We can use the composition mechanism in IOA to fix, for example, the size of the ring at 4. In Figure 5.4, the **type** statement explicitly identifies `abcd` as an enumerated type with four elements, and the **axioms** statement defines a ring structure on these four elements, which discharges the assumption in the definition of the single component.

```
type abcd = enumeration of a, b, c, d
axioms RingIndex(abcd, String)

automaton LCR4
  components theOnly: LCR(abcd)
```

Figure 5.4: IOA specification of four-process LCR algorithm

Even though the description of `LCR4` is not parameterized, it still defines a set of automata rather than single automaton: Figure 5.4 says nothing about how names are assigned to automata. We could pin down such details by creating and referring to an additional auxiliary specification, which defines the values of `name(a)`, `name(b)`, `name(c)`, and `name(d)`. But often it is not necessary to pin details down to such an extent, because the properties of an algorithm that are most of interest do not depend on these details.

5.3 Hiding output actions in a composition

IOA allows us to reclassify some (or all) of the output actions in a composite automaton as internal actions. Thus, for example, if we wish to hide the `send` and `receive` actions leading to the election of a leader in `LCR4`, we can use a **hidden** statement, as in Figure 5.5.

```
automaton LCR4a
  components theOnly: LCR4
  hidden receive(m, i, j), send(m, i, j)
```

Figure 5.5: IOA specification with hidden actions

6 IOA descriptions of properties of automata

IOA permits users to describe state invariants of I/O automata or simulation relations between I/O automata.

6.1 Invariants

Invariants are described using the keywords **invariant of** followed by the name of an automaton, a colon, and then a predicate. For example, the following invariant for the `LCR` automaton states that at most one process is ever elected as the leader.

```
invariant of LCR: P[i].status = elected ∧ P[j].status = elected ⇒ i = j
```

A state in a composite automaton is named by the name of the component to which it belongs followed by a dot followed by the state variable name, as shown in the invariant described above.[8]

6.2 Simulation relations

Simulation relations provide a convenient mechanism for showing that one automaton implements another, i.e., that every trace of one is a trace of the other. In order to illustrate various simulation relations, we describe a modification, `DelayedLossyChannel` (Figure 6.1), of the `LossyChannel` (Figure 4.3) automaton. In `DelayedLossyChannel`, the `crash` action does not result in the immediate loss of messages from the queue; rather, it marks messages as losable by subsequent internal `lose` actions.

```
axioms MarkedMessage for Mark[__]

automaton DelayedLossyChannel(M: type)
  signature
    input    insert(m: M), crash
    output   remove(m: M)
    internal lose
  states  buffer: Seq[Mark[M]] := {}
    transitions
      input insert(m)
        eff buffer := buffer ⊢ [m, false]
      output remove(m)
        pre buffer ≠ {} ∧ head(buffer).msg = m
        eff buffer := tail(buffer)
      input crash
        eff buffer := mark(buffer)
      internal lose
        eff buffer := choose
            ensuring subseqMarked(buffer', buffer)
```

Figure 6.1: Specification of an implementation of a lossy channel

The **axioms** statement in Figure 6.1 identifies a user-written specification (shown later in Figure 10.1) that defines a type constructor `Mark[__]` for types such as `Mark[M]` or `Mark[String]` of "marked messages." This specification defines a marked message to be a pair `[m, b]` of a message and a boolean value, the components of which can be extracted by the operators `.msg` and `.mark`. It also defines an operator `mark` that sets all marks in a sequence to `true`, an operator `messages` that given a sequence of marked messages returns the corresponding sequence of messages, and a relation `subseqMarked` that holds when the only messages missing from a sequence have marks of `true`.

[8] In the future, IOA may permit the name of the component to be omitted when there is no ambiguity (i.e., when only one component has a state variable with a given name).

The automaton DelayedLossyChannel implements the automaton LossyChannel, because all of its traces are also traces of LossyChannel. One way of showing that this is the case is to define a relation between the states of DelayedLossyChannel and those of LossyChannel and to show that this relation is a forward simulation (see Section 1.4). The following assertion in IOA defines such a relation.

```
forward simulation from DelayedLossyChannel to LossyChannel:
  messages(DelayedLossyChannel.buffer) = LossyChannel.buffer
```

It is also true that every trace of LossyChannel is a trace of DelayedLossyChannel, i.e., that the two automata have the same set of traces. One way to show this reverse inclusion is to define a relation between the states of LossyChannel and those of DelayedLossyChannel and to show that this relation is a backward simulation. The following assertion describes such a relation.

```
backward simulation from LossyChannel to DelayedLossyChannel:
  ∃ s: Seq[Mark[M]] (subseqMarked(s, DelayedLossyChannel.buffer)
                     ∧ LossyChannel.buffer = messages(s))
```

In order to establish that relations defined in these fashions are actually forward and backward simulation relations, the user must demonstrate that these relations satisfy the definitions given for simulation relations in Section 1.4. The key element in such a demonstration is usually the identification, for each step of one automaton, of an execution fragment of the other that contains the same external actions.

Editorial note: Need to add example of such an identification here, together with the formal syntax for describing identifications in the reference manual. In general, the identification is a definition by cases.

Part II
IOA Data Types

IOA specifications can employ various data types, both built-in and user-defined.

- The primitive data types `Bool`, `Char`, `Int`, `Nat`, `Real`, and `String` can be used without explicit declarations. Section 7 describes the operators available for each of these types.

- Other primitive data types can be introduced as **type** parameters to automaton definitions, as in the channel automaton described in Figure 2.2, which is parameterized by the types `M` and `Index`.

- Compound data types formed using the type constructors `Array`, `Map` `Mset`, `Null`, `Seq`, and `Set` can be used without explicit declarations. Section 8 describes the operators available for these types.

- Compound data types formed using the keywords **enumeration**, **tuple**, and **union** can be used with explicit declarations, as in
  ```
  type Color = enumeration of red, white, blue
  type Msg   = tuple of source, dest: Process, contents: String
  type Fig   = union of sq: Square, circ: Circle
  ```
  Sections 9.8 and 23 describe the operators available for these data types.

- User-defined data types, as well as additional operators on the above primitive and compound data types, can be introduced (or required to have certain properties) by indicating auxiliary specifications, as in
  ```
  axioms RingIndex(abcd, String)
  axioms Stack for Stack[__]
  assumes TotalOrdering(T, <)
  ```
  These auxiliary specifications, which users write as *traits* in the Larch Shared Language (LSL), provide both the syntax and semantics for all operators introduced in this fashion. Sections 9 and 10 describe how to write LSL traits and how to incorporate them into IOA specifications by means of the **axioms** statement.

In this section we describe the built-in types informally and list their operators. Appendix A defines the properties of these types and operators formally via sets of axioms in multisorted first-order logic (see Section 11). Data types and operators are defined abstractly, not in terms of any particular representation or implementation. In particular, operators are defined without any reference to a "state" or "store," so they cannot have "side-effects."

The equality (`__=__`), inequality (`__≠__`), and conditional (**if** `__` **then** `__` **else**) operators are available for all data types in IOA (the `__`'s are placeholders for the arguments of these operators).

7 Built-in primitive types

The following built-in primitive types and operators require no declaration.

7.1 Booleans

The boolean data type, `bool`, provides constants and operators for the set $\{true, false\}$ of logical values. Syntactically, the operator $\neg$ binds more tightly than the operators $\wedge$ and $\vee$, which bind more tightly than $\Rightarrow$, which binds more tightly than $\Leftrightarrow$. Appendix A.2 defines the semantics of this data type.

Operators for `bool`	Sample input	Meaning
`true`, `false`	`true`, `false`	The values *true* and *false*
$\neg$	`~p`	Negation (not)
$\wedge$, $\vee$	`p /\ q`, `p \/ q`	Conjunction (and), disjunction (or)
$\Rightarrow$	`p => q`	Implication (implies)
$\Leftrightarrow$	`p <=> q`	Logical equivalence (if and only if)

7.2 Integers

The integer data type, `Int`, provides constants and operators for the set of (positive and negative) integers. Appendix A.4 defines the semantics of this data type.

Operators for `Int`	Sample input	Meaning
`0`, `1`, ...	`123`	Non-negative integers
`-`	`-x`	Additive inverse (unary minus)
`abs`	`abs(x)`	Absolute value
`pred`, `succ`	`succ(x)`	Predecessor, successor
`+`, `-`, `*`	`x + (y*z)`	Addition, subtraction, multiplication
`min`, `max`	`min(x, y)`	Minimum, maximum
`div`, `mod`	`mod(x, y)`	Integer quotient, modulus
$<, \leq, >, \geq$	`x <= y`	Less (greater) than (or equal to)

Syntactically, all binary operators bind equally tightly, so that expressions must be parenthesized, as in `((x*y) + z) > 3`, to indicate the arguments to which operators are applied.[9]

7.3 Natural numbers

The natural number data type, `Nat`, provides constants and operators for the set of non-negative integers. The operators and constants are as for `Int`, except that there are no unary operators `-` or `abs`, there is an additional operator `**` for exponentiation, and the value of `x-y` is defined to be `0` if `x < y`. Appendix A.5 defines the semantics of this data type.

Syntactically, integer constants (e.g., `1`) and operators (e.g., `-`) are distinct from natural number constants and operators that have the same typographical representation. Sometimes such *overloaded* operators can be distinguished from context (e.g., the `1` in the expression `abs(-1)` must be an integer constant, because `abs` and unary `-` are operators over the integers, but not over the

[9] In the future, IOA may define a binding order for the symbols commonly used in arithmetic.

natural numbers). At other times, users must distinguish which operators or constants are meant by *qualifying* expressions with types, as in `x > 0:Nat`.

7.4 Real numbers

The real number data type, `Real`, provides constants and operators for the set of real numbers. Again, the operators and constants are as for `Int`, except that there are no operators `pred`, `succ`, `div`, and `mod`, and there are additional operators `/` and `**` for division and exponentiation. Appendix A.6 defines the semantics of this data type.

7.5 Characters

The character data type, `Char`, provides constants and operators for letters and digits.[10] Appendix A.3 defines the semantics of this data type.

Operators for `Char`	Sample input	Meaning
'A', ... , 'Z', 'a', ... , 'z', '0', ... , '9'	'J'	Letters and digits
$<, \leq, >, \geq$	'A' <= 'Z'	ASCII ordering

7.6 Strings

The string data type, `String`, provides constants and operators for lexicographically ordered sequences of characters. It provides operators as described for `Seq[Char]` (see Section 8.4) as well as the ordering relations $<$, $\leq$, $>$, and $\geq$. Appendix A.7 defines the semantics of this data type.

8 Built-in type constructors

The following built-in type constructors and operators require no declaration.

8.1 Arrays

For each $n > 0$, the array data type `Array[I1, ... , In, E]` provides constants and operators for `n`-dimensional arrays of elements of type `E` indexed by elements of types `I1`, ... , `In`. Appendix A.8 defines the semantics of this data type.

Operators for `Array[I1, ... In, E]`	Sample input	Meaning
constant	constant(e)	Array with all elements equal to `e`
__[__, ... , __]	a[i], b[i, j]	Element indexed by `i` (and `j`) in arrays `a`, `b`
assign	assign(a, i, e)	Array `a'`, equal to `a` except that `a'[i] = e`
	assign(b, i, j, e)	Array `b'` equal to `b` except that `b'[i, j] = e`

The dimension of the array denoted by `constant(e)` is determined by an explicit qualification, as in `constant(e):Array[I,I,E]`, or by context, as in `constant(e)[i]`.

[10] Additional character constants may be provided in a future version of IOA.

8.2 Finite sets

The set data type, `Set[E]`, provides constants and operators for finite sets of elements of type `E`. Appendix A.13 in defines the semantics of this data type.

Operators for `Set[E]`	Sample input	Meaning
`{}`	`{}`	Empty set
`{__}`	`{e}`	Set containing `e` alone
`insert`	`insert(e, s)`	Set containing `e` and all elements of `s`
`delete`	`delete(e, s)`	Set containing all elements of `s`, but not `e`
$\in$	`e \in s`	True iff `e` is in `s`
$\cup$, $\cap$, -	`(s \U s') - (s \I s')`	Union, intersection, difference
$\subset$, $\subseteq$, $\supset$, $\supseteq$	`s \subseteq s'`	(Proper) subset (superset)
`size`	`size(s)`	Size (an `Int`) of `s`

8.3 Finite multisets

The multiset data type, `Mset[E]`, provides constants and operators for finite multisets of elements of type `E`. Its operators are those for `Set[E]`, except that there is an additional operator `count` such that `count(e, s)` is the number (an `Int`) of times an element `e` occurs in a multiset `s`. Appendix A.10 defines the semantics of this data type.

8.4 Finite sequences

The sequence data type, `Seq[E]`, provides constants and operators for finite sequences of elements of type `E`. Appendix A.12 defines the semantics of this data type.

Operators for `Seq[E]`	Sample input	Meaning
`{}`	`{}`	Empty sequence
$\vdash$	`s |- e`	Sequence with `e` appended to `s`
$\dashv$	`e -| s`	Sequence with `e` prepended to `s`
$\|$	`s || s'`	Concatenation of `s`, `s'`
$\in$	`e \in s`	True iff `e` is in `s`
`head`, `last`	`head(s)`	First (last) element in sequence
`init`, `tail`	`tail(s)`	All but first (last) elements in sequence
`len`	`len(s)`	Length (an `Int`) of `s`
`__[__]`	`s[n]`	`n`th (an `Int`) element in `s`

8.5 Finite mappings

For each `n` $>$ 0, the mapping data type `Map[D1, ..., Dn, R]` provides constants and operators for finite partial mappings of elements of domain type `D1` $\times \cdots$ `Dn` to elements of range type `R`.

Appendix A.9 defines the semantics of this data type.

Finite mappings differ from arrays in that they may not be defined for all elements of `D`.

Operators for `Map[D1, ... Dn, R]`	Sample input	Meaning
`empty`	`empty`	Empty mapping
`__[__, ... , __]`	`m[d]`, `m2[d, e]`	Image of `d` (and `e`) under `m`, `m2`
`defined`	`defined(m, d)`	True if `m[d]` is defined
`update`	`update(m, d, r)`	Mapping `m′` equal to `m` except that `m′[d] = r`

8.6 Null elements

The data type `Null[E]` contains a copy of each element of the underlying data type `E`, plus one additional element `nil`. Appendix A.11 defines the semantics of this data type.

Operators for `Null[E]`	Sample input	Meaning
`nil`	`nil`	Undefined element
`embed`	`embed(e)`	The element `e`
`.val`	`n.val`	Either `nil` or an `e` such that `n = embed(e)`

9 Data type semantics

IOA describes the semantics of abstract data types by means of axioms expressed in the the Larch Shared Language (LSL). Users need refer to LSL specifications only if they have questions about the precise mathematical meaning of some operator or if they wish to introduce new operators or data types.[11]

This section provides a tutorial introduction to LSL. It is taken from Chapter 4 of [8], but has been updated to reflect several changes to LSL, most significantly the addition of explicit quantification. LSL is a member of the Larch family of specification languages [8], which supports a two-tiered, definitional style of specification. Each specification has components written in two languages: LSL, which is independent of any programming language, and a so-called *interface language* tailored specifically for a programming language (as LCL is tailored for C) or for a mathematical model of computation (as IOA is tailored for I/O automata). Interface languages are used to specify interfaces between program components and the effects of executing those components. By tailoring interface languages to programming languages or mathematical models, Larch makes it easy to describe the details of an interface (e.g., how program modules communicate) in a fashion that is familiar to users.

Interface languages rely on definitions from auxiliary specifications, written in LSL, to provide semantics for the data types a program manipulates. An LSL specification, known as a *trait*, describes a collection of *sorts* (i.e., non-empty sets of elements) and *operators* (i.e., functions mapping tuples of elements to elements), by means of axioms written in first-order logic. For example, the `Sequences` trait shown in Figure 9.1 describes some properties of finite sequences of elements of

[11]Some tool builders may wish to provide other, equivalent definitions for the built-in data types, e.g., using some other mathematical formalism or in terms of procedures written in some programming language.

```
Sequences(E): trait
  introduces
    {}:                 → Seq[E]
    __⊣__: E, Seq[E] → Seq[E]
    last:  Seq[E]     → E
    init:  Seq[E]     → Seq[E]
  asserts with s: Seq[E], e: E
    sort Seq[E] generated freely by {}, ⊣;
    last(e ⊣ s) = (if s = {} then e else last(s));
    init(e ⊣ s) = (if s = {} then {} else e ⊣ init(s));
  implies with s1, s2: Seq[E], e1, e2: E
    e1 ⊣ s1 = e2 ⊣ s2 ⇔ e1 = e2 ∧ s1 = s2;
    e1 ⊣ s1 ≠ {}
```

Figure 9.1: Simplified LSL specification for sequences

a sort `E`. The **introduces** clause lists the sorts and operators being specified, the **asserts** clause defines their properties, and the **implies** clause calls attention to some (purported) consequences of these properties. In the **introduces** clause, the `__`'s are placeholders for the arguments of the infix operator ⊣. In the **asserts** clause, the **generated freely by** axiom asserts that all sequences can be obtained by prepending a finite number of elements (using the operator ⊣) to the empty sequence `{}`, and the remaining axioms provide inductive definitions of the `last` and `init` operators; note that `last({})` and `init({})` are not defined. The **implies** clause calls attention to the fact that two elements of the freely generated sort `Seq[E]` are equal if and only if they were generated in the same fashion; this fact distinguishes sequences from sets, where the order in which elements are inserted does not matter.

Larch distinguishes the idealized sorts of elements described in LSL (such as arbitrarily long sequences) from the actual types of elements involved in a computation (such as sequences of some limited length). Larch also distinguishes between mathematical operations on sorts (such as `last`, which is not specified completely) and computational procedures (such as one that returns the first element in a sequence, which may either return an "error" element or raise an exception if the sequence is empty). Each data type in an IOA program is interpreted as a sort in LSL, and the results of computations are specified in terms of operators whose meanings have been defined in LSL.

9.1 Axiomatic specifications

LSL's basic unit of specification is a *trait*. Consider, for example, the specification for some properties of sets given in Figure 9.2. This specification is similar to conventional algebraic specifications, as would be written in many languages [1, 3]. The trait has a name, `Set0`, which is independent of the names appearing in it for data abstractions (e.g., `Set[E]`) or for operators (e.g., $\in$).

The part of the trait following the keyword **introduces** declares a list of *operators*, each with its *signature* (the *sorts* of its *domain* and *range*). An operator is a total function that maps a tuple of values of its domain sorts to a value of its range sort. It is important to note that an operator in LSL is always a total function even if its value is not specified for every element of its domain. For example, `{}` is in the domain of the function `last` in Figure 9.1, even though no value is specified for `last({})`. Nonetheless, `last({})` denotes a fixed element of sort `E`.

Every operator used in a trait must be declared; signatures are used to sort-check *terms* in

```
Set0: trait
  introduces
    {}:                 → Set[E]
    insert: E, Set[E] → Set[E]
    __∈__:  E, Set[E] → Bool
    size:   Set[E]    → Int
    0, 1:             → Int
    __+__:  Int, Int  → Int
    __≥__:  Int, Int  → Bool
  asserts with s, s′: Set[E], e, e′: E
    ∀ e (e ∈ s ⇔ e ∈ s′) ⇒ s = s′;
    ¬(e ∈ {});
    e ∈ insert(e′, s) ⇔ e = e′ ∨ e ∈ s;
    size({}) = 0;
    size(insert(e, s)) = size(s) + (if e ∈ s then 0 else 1)
```

Figure 9.2: A trait specifying some properties of sets

much the same way as expressions are type-checked in programming languages. Primitive sorts are denoted by identifiers (such as `E` and `Int`); sorts constructed from other sorts (in a manner defined by the trait) are denoted by identifiers for sort constructors (such as `Set`) applied to the other sorts (as in `Set[E]`). All sorts are declared implicitly by their appearance in signatures.

Double underscores (`__`) in an operator declaration indicate that the operator will be used in *mixfix terms*. For example, ∈, `+`, and ≥ are declared as binary infix operators. Infix, prefix, postfix, and bracketing operators (such as `__+__`, `-__`, `__!`, `{__}`, `__[__]`, and **if** `__` **then** `__` **else** `__`) are integral parts of many familiar mathematical and programming notations, and their use can contribute substantially to the readability of specifications.

LSL's grammar for mixfix terms is intended to ensure that legal terms parse as readers expect—even without studying the grammar. LSL has a simple precedence scheme for operators:

- postfix operators that consist of a dot followed by an identifier (as in field selectors such as `.first`) bind most tightly;
- bracketing operators that begin with a left delimiter (e.g., `[`) and end with a right delimiter (e.g., `]`) bind more tightly than
- the logical quantifiers ∀ (*for all*) and ∃ (*there exists*), which bind more tightly than
- other user-defined operators and the built-in propositional operator ¬ (*not*), which bind more tightly than
- the built-in equality and inequality operators $=$ and $\neq$ which bind more tightly than
- the built-in propositional operators ∧ (*and*) and ∨ (*or*), which bind more tightly than
- the built-in propositional operator ⇒(*implies*), which binds more tightly than
- the built-in propositional operator ⇔ (*if and only if*), which binds more tightly than
- the built-in conditional operator **if** `__` **then** `__` **else** `__`.

For example, the term `p ⇔ x + w.a.b = y ∨ z` can be written without parentheses and is equivalent to the fully parenthesized term `p ⇔ (((x + ((w.a).b)) = y) ∨ z)`. LSL allows unparenthesized infix terms with multiple occurrences of an operator at the same precedence level, but not different operators; it associates such terms from left to right. Fully parenthesized terms are always acceptable. Thus `x ∧ y ∧ z` is equivalent to `(x ∧ y) ∧ z`, but `x ∨ y ∧ z` must be written as `(x ∨ y) ∧ z` or as `x ∨ (y ∧ z)`, depending on which is meant.

The part of the trait following the keyword **asserts** constrains the operators by means of *formulas*, that is, by terms of sort `bool` constructed from variables declared following the keyword **with**, operators declared in the trait, built-in operators, and quantifiers. The last three formulas in the trait `Set0` are *equations*, which consist of two quantifier-free terms of the same sort, separated by = or ⇔.

Each trait defines a *theory* (a set of formulas) in multisorted first-order logic (see Section 11). Each theory contains the trait's assertions, the conventional axioms of first-order logic, everything that follows from them, and nothing else. This *loose* semantic interpretation guarantees that formulas in the theory follow only from the presence of assertions in the trait—never from their absence. This is in contrast to algebraic specification languages based on initial algebras [7] or final algebras [15]. Using the loose interpretation ensures that all theorems proved about an incomplete specification remain valid when it is extended.

Each trait should be *consistent*: it must not define a theory containing the formula `false`. Consistency is often difficult to prove and is undecidable in general. Inconsistency is often easier to detect and can be a useful indication that there is something wrong with a trait.

9.2 Axiom schemes

At times, it can be difficult to find adequate sets of axioms that assert some property of interest. Consider, for example, the problem of asserting that the set `Nat` of natural numbers contains the integers 0, 1, 2, ... and nothing else. A natural approach is to assert that the set `Nat` is the smallest set containing 0 and closed under the successor operation `succ` (defined by `succ(n) = n+1`):

```
∀ s:Set[Nat] (0 ∈ s ∧ ∀ n:Nat (n ∈ s ⇒ succ(n) ∈ s) ⇒ ∀ n:Nat (n ∈ s))
```

However, the axioms in the trait `Set0` do not imply the existence of enough elements of sort `Set[E]` to give this assertion its intended meaning: these axioms remain true if `Set[E]` is interpreted as containing only finite sets of elements of sort `E`, in which case no element of `Set[Nat]` is closed under `succ` and the assertion about `Nat` is vacuously true.

There are several ways to remedy this problem. One is to posit some special, unaxiomatized relationship between the sort `Set[E]` and the sort `E` (i.e., that `Set[E]` contains *all* sets of elements of `E`). However, this approach creates another problem, namely, whether to posit other special relationships between similar notations such as `Seq[E]` or `Map[E,E]` and the sort `E`. Another approach, which avoids this problem, is to enlarge `Set0` with axioms like `∃ s:Set[Nat] ∀ n:Nat (n ∈ s)` that force `Set[E]` to contain sufficiently many sets of elements of `E`. Unfortunately, no finite set of axioms suffices to force the existence of all potentially interesting sets of elements of `E`.

For reasons such as this, LSL provides another statement, the **generated by** statement, for use in defining theories that would otherwise require infinitely many axioms. A **generated by** statement (such as the first axiom in the trait `Sequences`) asserts that a list of operators is a complete set of *generators* for a sort. That is, each value of the sort is equal to one that can be described using just those operators. For example, the statement

```
    sort Nat generated freely by 0, succ
```

asserts that all values of sort `Nat` can be constructed by finitely many applications of the operator `succ` to the constant `0`. In addition, the keyword **freely** indicates that the generators for `Nat` provide unique representations for the natural numbers (i.e., that `succ(n)` $\neq$ `0` and $\mathtt{succ}^i\mathtt{(0)} = \mathtt{succ}^j\mathtt{(0)}$ if and only if $i = j$). Similary, the statement

```
sort Set[E] generated by {}, insert
```

asserts that all values of sort `Set[E]` can be constructed by finitely many applications of `insert` to `{}`, that is, that all values of sort `Set[E]` are finite sets. In this case, the absence of the keyword **freely** suggests that the generators for `Set[E]` do not provide unique representations for sets of elements of `E`; in fact, the axioms for sets ensure that `insert(a, insert(b, {})) = insert(b, insert(a, {}))`.

A **generated by** statement justifies an *induction schema* for proving properties of a sort. For example, to prove ∀ `s:Set[E] (size(s)` $\geq$ `0)` from the axioms of `Set0` and the **generated by** statement for `Set[E]`, we could (try to) construct a *proof by induction* with the structure

- Basis step: `size({})` $\geq$ `0`
- Induction step: ∀ `s:Set[E]` ∀ `e:E (size(s)` $\geq$ `0` $\Rightarrow$ `size(insert(e, s))` $\geq$ `0)`

In general, a **generated by** statement is equivalent to an infinite set of formulas, one for each property (such as `size(s)` $\geq$ `0`) that can be expressed in first-order logic.[12]

9.3 Combining LSL specifications

The trait `Set0` contains four operators that it does not define: `0`, `1`, `+`, and $\geq$. Without more information about these operators, the definition of `size` is not particularly useful, and we cannot prove "obvious" properties such as `size(s)` $\geq$ `0`. We could add assertions to `Set0` to define these operators, but it is usually better to specify such operators in a separate trait that is included by reference. This makes the specification more structured and makes it easier to reuse existing specifications. Hence we might remove the explicit introductions of these operators from `Set0` and instead add an *external reference*

```
includes Integer
```

to a separate trait `Integer` (see Appendix A.4), which both introduces these operators and defines their properties.

The theory associated with a trait containing an **includes** clause is the theory associated with the assertions of that trait and all (transitively) included traits.

It is often convenient to combine several traits dealing with different aspects of the same operator. This is common when specifying something that is not easily thought of as a data type. For example, the trait `PartialOrder1` is equivalent to the less structured trait `PartialOrder2` in Figure 9.3. Both traits define a partial order to be an irreflexive, transitive order, but `PartialOrder1` makes this characterization of the definition explicit.

9.4 Renaming sorts and operators in LSL specifications

The trait `PartialOrder1` relies heavily on the use of the same operator symbol, <, and the same sort identifier, `T`, in the two included traits. In the absence of such happy coincidences, renaming can be used to make names coincide, to keep them from coinciding, or simply to replace them with more suitable names, as in

[12]LSL provides an additional axiom scheme in the form of a **partitioned by** statement, which asserts that a list of operators is a complete set of *observers* for a sort: all distinct values of the sort can be distinguished using just these operators. For example, the statement **sort** `Set[E]` **partitioned by** ∈ asserts that terms indistinguishable by the observer ∈ denote the same value of sort `Set[E]`. This statement is equivalent to the first axiom in the trait `Set0`. In general, **partitioned by** statements do not increase the descriptive power of LSL, because they can be reformulated as single axioms that contain explicit quantifiers. However, they can be used to provide proof tools with automatic methods of deduction.

```
Irreflexive: trait
  introduces __<__: T, T → Bool
  asserts with x: T
    ¬(x < x)

Transitive: trait
  introduces __<__: T, T → Bool
  asserts with x, y, z: T
    x < y ∧ y < z ⇒ x < z

PartialOrder1: trait
  includes Irreflexive, Transitive

PartialOrder2: trait
  introduces __<__: T, T → Bool
  asserts with x, y, z: T
    ¬(x < x);
    x < y ∧ y < z ⇒ x < z
```

Figure 9.3: Specifications of kinds of relations

```
    includes Transitive(⊂ for <)
```

which we can use to assert that the operator ⊂ is transitive.

In general, a *trait reference* is a phrase *Tr*(*name1* **for** *name2*, ...) that stands for the trait *Tr* with every occurrence of *name2* (which must be a sort, a sort constructor, or an operator) replaced by *name1*, etc. If *name2* is a sort or a sort constructor, this renaming changes the signatures of all operators in *Tr* in whose signatures *name2* appears. For example, the signature of the operator ⊣ changes to `Int,Seq[Int]→Seq[Int]` in the trait reference **includes** `Sequences(Int` **for** `E)`.

Any sort or operator in a trait can be renamed when that trait is referenced in another trait. Some, however, are more likely to be renamed than others. It is often convenient to single these out so that they can be renamed positionally. For example, the header `Sequences(E):` **trait** in Figure 9.1 makes the reference **includes** `Sequences(Int)` equivalent to **includes** `Sequences(Int` **for** `E)`.

9.5 Stating intended consequences of LSL specifications

It is not possible to prove the "correctness" of a specification, because there is no absolute standard against which to judge correctness. But since specifications can contain errors, specifiers need help in locating them. LSL specifications cannot, in general, be executed, so they cannot be tested in the way that programs are commonly tested. LSL sacrifices executability in favor of brevity, clarity, flexibility, generality, and abstraction. To compensate, it provides other ways to check specifications.

This section briefly describes ways in which specifications can be augmented with redundant information to be checked during validation. Checkable properties of LSL specifications fall into three categories: *consistency*, *theory containment*, and *completeness*. As discussed earlier, the requirement of consistency means that any trait whose theory contains the formula `false` is illegal.

An **implies** clause makes claims about theory containment. Suppose we think that a consequence of the assertions of `Set0` is that the order in which elements are inserted in a set makes no difference. To formalize this claim, we could the following clause to `Set0`:

```
    implies with e1, e2: E, s: Set[E]
```

```
insert(e1, insert(e2, s)) = insert(e2, insert(e1, s))
```

Properties claimed to be implied can be specified using the full power of LSL, including formulas, **generated by** statements, and references to other traits. Attempting to verify that properties are actually implied can be helpful in error detection. Implications also help readers confirm their understanding. Finally, they can provide useful lemmas that will simplify reasoning about specifications that use the trait.

LSL does not require that each trait define a *complete theory*, that is, one in which each fully quantified formula is either true or false. Many finished specifications (intentionally) do not fully define all their operators. Furthermore, it can be useful to check the completeness of some definitions long before finishing the specification of which they are part. Therefore, instead of building in a single test of completeness that is applied to all traits, LSL provides a way to include within a trait specific checkable claims about completeness, using **converts** clauses. Adding the clause

```
implies converts ∈
```

to `Set0` makes the claim that the trait's axioms fully define the operator $\in$. This claim means that, if the interpretations of all the other operators are fixed, there is only one interpretation of $\in$ that satisfies the axioms. (This claim cannot be proved from the axioms in `Set0` alone, but can be proved from those axioms together with the induction schema associated with **sort** `Set[E]` **generated by** `{}, insert`.)

The claim **implies converts** `last, init` cannot be verified from the axioms for `Sequences` in Figure 9.1, which define the meaning of `last(s)` and `init(s)` only when `s` $\neq$ `{}`. This incompleteness in `Sequences` can be resolved by adding other axioms to the trait, perhaps `last({})` $=$ `errorVal`. But it is generally better not to add such axioms. The specifier of `Sequences` should not be concerned with whether the sort `E` has an `errorVal` and should not be required to introduce irrelevant constraints on `__⊣__`. Extra axioms give readers more details to assimilate; they may preclude useful specializations of a general specification; and sometimes there simply is no reasonable axiom that would make an operator convertible (consider division by 0). Error conditions and undefined values are treated best in interface specifications, as discussed below.

LSL provides an **exempting** clause for listing terms that are not claimed to be defined (which is different from "that are claimed to be undefined"). The claim

```
implies with d: D
  converts last, init exempting last({}), init({})
```

means that `last` and `init` are fully defined by the trait's axioms, interpretations for the other operators (`{}` and ⊣), and interpretations for the two terms `last({})` and `init({})`. This claim can be proved by induction from the axioms of `Sequences`.

In IOA, the use of incompletely defined terms results in underspecification, not nondeterminism. Consider, for example, the sample automaton `Choice` (Figure 4.1), in which the statement

```
num: Int := choose n where 1 ≤ n ∧ n ≤ 3
```

illustrated the nondeterministic choice of an initial value for the state variable `num`. If we replace that statement by

```
num: Int := someElement(1, 3)
```

where `someElement` is specified by the trait

```
SomeElement: trait
  includes Integer
  introduces someElement: Int, Int → Int
  asserts with m, n: Int
    m ≤ n ⇒ m ≤ someElement(m, n) ∧ someElement(m, n) ≤ n
```

then the initial value of `num` is no longer nondeterministic: it is always equal to the value of `someElement(1, 3)`. The trait `SomeElement` does not tell us *which* value this is, but it is always the *same* value, because `someElement` denotes a fixed function, not a method of computation.

9.6 Recording assumptions in LSL specifications

Some traits are suitable for use in all contexts and some only in certain contexts. Just as we write preconditions that describe the contexts in which a procedure may be called, we write *assumptions* in traits that describe the contexts in which the traits may be included. As with preconditions, assumptions impose proof obligations on the client (i.e., the including trait), and they may be presumed true within the included trait.

Consider, for example, specializing the `Sequences` trait to describe sequences of strings by combining `Sequences` with a separate trait that defines operators for the data type `String`:

```
StringSequences: trait
  includes Sequences(String), String
```

The interactions between `String` and `Sequences` are limited. Nothing in `Sequences(String)` depends on any particular operators being introduced in including traits, let alone their having any special properties. Therefore `Sequences` needs no assumptions.

```
OrderedSequences0(E): trait
  includes Sequences
  introduces
    __<__:  E, E           → Bool
    __≪__: Seq[E], Seq[E] → Bool
  asserts with s, s1, s2: Seq[E], e, e1, e2: E
    {} ≪ (e ⊣ s);
    ¬(s ≪ {});
    (e1 ⊣ s1) ≪ (e2 ⊣ s2) ⇔ e1 < e2 ∨ (e1 = e2 ∧ s1 ≪ s2)
```

Figure 9.4: Preliminary specification of ordered sequences

Consider, however, specializing the `Sequences` trait to describe lexicographically ordered sequences, as in Figure 9.4. As written, `OrderedSequences0` says nothing about whether the operator $<$ defines an ordering over `E`; hence there is no reason to believe that the operator $\ll$ defines an ordering over `Seq[E]`. It is inappropriate to define $<$ within `OrderedSequences0`, both because its definition would depend on properties of the sort `E` (which are not specified in `OrderedSequences0`) and because to define $<$ there would overly restrict the utility of `OrderedSequences0`. What we need is an **assumes** clause, as in the trait `OrderedSequences` in Figure 9.5.

```
OrderedSequences(E): trait
  assumes Transitive(E for T)
  includes Sequences
  introduces
    __<__:  E, E           → Bool
    __≪__: Seq[E], Seq[E] → Bool
  asserts with s, s1, s2: Seq[E], e, e1, e2: E
    {} ≪ (e ⊣ s);
    ¬(s ≪ {});
    (e1 ⊣ s1) ≪ (e2 ⊣ s2) ⇔ e1 < e2 ∨ (e1 = e2 ∧ s1 ≪ s2)
  implies trait Transitive(Seq[E] for T, ≪ for <)
```

Figure 9.5: Specification of ordered sequences

Since `OrderedSequences` may presume its assumptions, its theory is the same as if it had included `Transitive` rather than assumed it: `OrderedSequences` inherits all the declarations and assertions of `Transitive`. Therefore, the assumption of `Transitive` can be used to derive various properties of `OrderedSequences`, for example, that $\ll$ is itself transitive, as claimed in the **implies** clause.

The difference between **assumes** and **includes** appears when `OrderedSequences` is used in another trait. Whenever a trait with assumptions is included or assumed, its assumptions must be *discharged*. For example, in

```
StringSequences1: trait
  includes String, OrderedSequences(String)
```

the assumption to be discharged is that the (renamed) theory associated with `Transitive` is a subset of the theory associated with the rest of `StringSequences1` (i.e., that it is a subset of the theory associated with the trait `String`).

9.7 Built-in operators, overloading, and qualification

In our examples, we have freely used the propositional operators together with three heavily overloaded operators, **if** __ **then** __ **else** __, =, and ≠, which are built into LSL. This allows these operators to have appropriate syntactic precedence. More importantly, it guarantees that they have consistent meanings in all LSL specifications, so readers can rely on their intuitions about them.

Similarly, LSL can recognize decimal numerals, such as 0, 24, and 1997, without explicit declarations and definitions. In principle, each numeral could be defined within LSL, but such definitions are not likely to advance anyone's understanding of the specification. `DecimalLiterals` is a predefined quasi-trait that implicitly defines all the numerals that appear in a specification; it is included in the standard numeric traits `Natural`, `Integer`, and `Real` that are built into IOA (see Appendix A).

In addition to the built-in overloaded operators and numerals, LSL provides for user-defined overloadings. Each operator must be declared in an **introduces** clause and consists of an identifier (e.g., `update`) or operator symbol (e.g., `__<__`) and a signature. The signatures of most occurrences of overloaded operators are deducible from context. Consider, for example, the trait `OrderedSequences(<` **for** $\ll$`)`, in which the symbol < denotes two different operators, one relating terms of sort `E`, and the other, terms of sort `Seq[E]`. The contexts in which this symbol is used determine unambiguously which operator is which.

LSL provides notations for disambiguating overloaded operators when context does not suffice. Any subterm of a term can be qualified by its sort. For example, `a:S` in `a:S = b` explicitly indicates that `a` is of sort `S`. Furthermore, since the two operands of = must have the same sort, this qualification also implicitly defines the signatures of = and `b`. These notations can be used to disambiguate the overloaded operator symbol < in the last axiom in `OrderedSequences(<` **for** $\ll$`)` explicitly, as in

```
(e1 ⊣ s1):Seq[E] < (e2 ⊣ s2):Seq[E] ⇔
  e1:E < e2:E ∨ (e1:E = e2:E ∧ s1:Seq[E] < s2:Seq[E])
  t1:T < t2:T ∨ (t1:T = t2:t ∧ s1:Seq[T] < s2:Seq[T])
```

In contexts other than terms, overloaded operators can be disambiguated by directly affixing their signatures, as in **implies converts** `<:Seq[E],Seq[E]→Bool`.

9.8 Shorthands

Enumerations, tuples, and unions provide compact, readable representations for common kinds of theories. They are syntactic shorthands for things that could be written in LSL without them.

Enumerations

The enumeration shorthand defines a finite ordered set of distinct constants and an operator that enumerates them. For example,

```
    Status enumeration of waiting, elected, announced
```

is equivalent to including a trait with the body appearing in Figure 9.6.

```
SampleEnumeration: trait
  introduces
    waiting, elected, announced:        → Status
    succ:                        Status → Status
  asserts
    sort Status generated freely by waiting, elected, announced;
    succ(waiting) = elected;
    succ(elected) = announced
```

Figure 9.6: Expansion of an enumeration shorthand

Tuples

The tuple shorthand is used to introduce fixed-length tuples, similar to records in many programming languages. For example,

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

is equivalent to including a trait with the body appearing in Figure 9.7. Each field name (e.g., `source`) is incorporated in two distinct operators (e.g., `__.source` and `set_source`).

```
SampleTuple: trait
  introduces
    [__, __, __]: Message, Node, Set[Node] → Packet
    __.contents:  Packet                   → Message
    __.source:    Packet                   → Node
    __.dest:      Packet                   → Set[Node]
    set_contents: Packet, Message          → Packet
    set_source:   Packet, Node             → Packet
    set_dest:     Packet, Set[Node]        → Packet
  asserts with m, m1: Message, n, n1: Node, s, s1: Set[Node]
    sort Packet generated freely by [__, __, __];
    sort Packet partitioned by .contents, .source, .dest;
    [m, n, s].contents = m;
    [m, n, s].source = n;
    [m, n, s].dest = s;
    set_contents([m, n, s], m1) = [m1, n, s];
    set_source([m, n, s], n1) = [m, n1, s];
    set_dest([m, n, s], s1) = [m, n, s1]
```

Figure 9.7: Expansion of a tuple shorthand

Unions

The union shorthand corresponds to the tagged unions found in many programming languages. For example,

```
Figure union of sq: Square, circ: Circle
```
is equivalent to including a trait with the body appearing in Figure 9.8. Each field name (e.g., `circ`) is incorporated in three distinct operators (e.g., `circ:→Figure_tag`, `circ:Circle→Figure`, and `__.circ:Figure→Circle`).

```
SampleUnion: trait
  Figure_tag enumeration of sq, circ
  introduces
    sq:       Square → Figure
    circ:     Circle → Figure
    __.sq:    Figure → Square
    __.circ:  Figure → Circle
    tag:      Figure → Figure_tag
  asserts with s: Square, c: Circle
    sort Figure generated freely by sq, circ;
    sort Figure partitioned by tag, .sq, .circ;
    tag(sq(s)) = sq;
    tag(circ(c)) = circ;
    sq(s).sq = s;
    circ(c).circ = c
```

Figure 9.8: Expansion of a union shorthand

10 User-defined data types in IOA

Users of IOA can define additional data types and type constructors, define additional operators for the built-in data types or type constructors, or completely redefine the built-in data types or type constructors, by providing sets of axioms (as described in Section 9) for the new data types and operators.

Defining new data types To define and use a new abstract data type, one writes axioms for the data type in LSL and incorporates these axioms into an IOA specification using either an **axioms** or an **assumes** statement. For example, the index data type used in the leader election example (Section 5.1) is defined by the axioms in the trait `RingIndex` (Figure 5.2). This trait provides notations for two sorts (`I` and `J`) and four operators

```
first:        → I
left, right: I → I
name:        I → J
```

It also provides five axioms that constrain the properties of these operators (e.g., by requiring that different elements of type `I` have different names). However, it does not completely define these operators (e.g., it does not provide any concrete representation for the elements of type `J`).

The statement **axioms** `RingIndex(abcd, String)` appearing before the definition of the automaton `LCR4` (Figure 5.4) instantiates the parameters `I` and `J` in the trait `RingIndex` by the actual types `abcd` and `String`, thereby introducing notations for the operators

```
first:           → abcd
left, right: abcd → abcd
name:        abcd → String
```

and five axioms that define their properties. Again, the axioms do not completely define the operators; for example, they do not specify which element of `abcd` is the first (it need not be `a`), and they do not specify which strings are used to name the elements of `abcd`. When reasoning about `LCR4`, one can rely only on the properties of the operators given by the trait `RingIndex`.

As in LSL (see Section 9.6), the statement **assumes** `RingIndex(I, String)` appearing in the definitions of the automata `Process` (Figure 5.1) and `LCR` (Figure 5.3) both provides (and defines) notations for use in the definitions of those automata and also imposes proof obligations that must be discharged whenever they are used as components of other automata. When `Process` is used as a component of `LCR`, the **assumes** statement in the definition of `LCR` discharges this obligation by repeating the assumption contained in the definition of `Process`. When `LCR` is used as a component of `LCR4`, the **axioms** statement cited above discharges this proof obligation by defining the type `abcd` to have the required properties.

Defining new type constructors The statement **axioms** `MarkedMessage` **for** `Mark[__]` appearing before the definition of the automaton `DelayedLossyChannel` (Figure 6.1) enables IOA to recognize types such as `Mark[M]` in that definition, and it provides notations and axioms for operators such as `.msg` and `mark` appearing in that definition. These notations and axioms are found in the trait `MarkedMessage` (Figure 10.1), which has a single type parameter corresponding to the placeholder `__` for the single argument of the type constructor `Mark`.

```
MarkedMessage(M): trait
  Mark[M] tuple of msg: M, mark: Bool
  includes Sequence(Mark[M]), Sequence(M)
  introduces
    mark:          Seq[Mark[M]]               → Seq[Mark[M]]
    messages:      Seq[Mark[M]]               → Seq[M]
    subseqMarked:  Seq[Mark[M]], Seq[Mark[M]] → Bool
  asserts with mm, mm1, mm2: Mark[M], mms, mms1, mms2: Seq[Mark[M]]
    mark({}) = {};
    mark(mms ⊢ mm) = mark(mms) ⊢ [mm.msg, true];
    messages({}) = {};
    messages(mms ⊢ mm) = messages(mms) ⊢ mm.msg;
    subseqMarked(mms, {}) ⇔ mms = {};
    subseqMarked({}, mms ⊢ mm) ⇔ subseqMarked({}, mms) ∧ mm.mark;
    subseqMarked(mms1 ⊢ mm1, mms2 ⊢ mm2) ⇔
      (subseqMarked(mms1 ⊢ mm1, mms2) ∧ mm2.mark) ∨
      (subseqMarked(mms1, mms2) ∧ mm1 = mm2)
  implies with m: M, mms, mms1, mms2, mms3: Seq[Mark[M]]
    subseqMarked(mms, mms);
    subseqMarked(mms, mms ⊢ [m, true]);
    (subseqMarked(mms1, mms2) ∧ subseqMarked(mms2, mms3))
      ⇒ subseqMarked(mms1, mms3);
```

Figure 10.1: Definition of type constructor `Mark[__]`

Redefining built-in type constructors The statement **axioms** `Subseqence` **for** `Seq[__]` appearing before the definition of the automaton `LossyChannel` (Figure 4.3) overrides the built-in definition of the type constructor `Seq[__]`. Ordinarily, axioms for that type constructor are obtained from a built-in trait `Sequence(E)` (see Appendix A.12). In the presence of this **axioms** statement, axioms for `Seq[__]` are obtained instead from the trait `Subsequence`. Since `Subsequence` includes `Sequence`, the new definition actually extends the old: it introduces a single new operator, $\preceq$, and defines its properties.

Part III
IOA Reference Manual

This reference manual describes the syntax, static semantics, and logical semantics both of IOA specifications and of assertions about IOA specifications. The *syntax* for IOA describes, using a context-free (BNF) grammar, the notations that appear in IOA specifications and assertions. *Static semantics* impose restrictions on the notations allowed by this BNF grammar. A static checker can be used to detect when these restrictions are violated. The *logical semantics* for IOA describes, in mathematical terms, the meaning of specifications and assertions. Proof tools can provide assistance in checking assertions.

11 Logical preliminaries

The logical semantics of IOA (and LSL) are formalized in *multisorted first-order logic*, which serves to model precise mathematical usage. This section provides a brief, abstract overview of first-order logic.

11.1 Syntax

We start by describing an abstract syntax for mathematical expressions, that is, for expressions in multisorted first-order logic.

A *vocabularly* $\mathcal{V}$ for first-order logic is a set of *symbols* that denote two kinds of objects: *sorts*, denoted by symbols in $\mathcal{V}_{sorts}$, and *operators*, denoted by symbols in $\mathcal{V}_{ops}$.[13] In IOA and LSL, symbols such as `Bool`, `Set[Int]`, and `T` denote sorts, and symbols such as `0:→Int`, `__+__:Int,Int→Int`, `f:T→T`, and `__≠__:S,S→Bool` denote operators.

$\mathcal{V}^*_{sorts}$ is the set of all finite sequences of elements of $\mathcal{V}_{sorts}$, including the zero-length sequence.

The set $\mathcal{V}_{sigs}$ of *signatures* for a vocabulary $\mathcal{V}$ is the set of all pairs $\langle domain, range\rangle$ in which $domain \in \mathcal{V}^*_{sorts}$ and $range \in \mathcal{V}_{sorts}$.

Associated with each operator, *op*, in a vocabulary $\mathcal{V}$ is an *identifier*, *op.id*, and a signature, *op.sig*, in $\mathcal{V}_{sigs}$. For example, in IOA and LSL, `0`, `+`, `f`, and $\neq$ are operator identifiers and `→Int`, `Int,Int→Int`, `T→T`, and `S,S→Bool` are signatures (the sequence of sort symbols preceding the → constitutes the domain, and the sort symbol following the → is the range). The *arity* of an operator is the number of sort symbols in its domain. A *constant* is an operator of arity 0.

In general, first-order logic restricts attention to vocabularies $\mathcal{V}$ that contain the sort symbol `Bool`, the 0-ary operators `true` and `false` with signature `→Bool`, the unary operator $\neg$ with signature `Bool→Bool`, and the binary operators $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$ with signature `Bool,Bool→Bool`. First-order logic *with equality* in addition restricts attention to vocabularies $\mathcal{V}$ that contain, for every sort `S` in $\mathcal{V}_{sorts}$, the binary operators $=$ and $\neq$ with signature `S,S→Bool`. For IOA and LSL, we also restrict attention to vocabularies $\mathcal{V}$ that contain, for every sort `S` in $\mathcal{V}_{sorts}$, the conditional operation `if__then__else__` with signature `Bool,S,S→Bool`.

A *variable* is a symbol, v, with which is associated an identifier, $v.id$, and a sort, $v.sort$; v is a *variable over* $\mathcal{V}$ if $v.sort$ is in $\mathcal{V}_{sorts}$. In IOA and LSL, symbols such as `n:Int` and `x:Set[Int]` are variables.

[13]A logic in which $\mathcal{V}_{sorts}$ contains more than one symbol is called *multisorted*.

For any vocabulary $\mathcal{V}$, a *$\mathcal{V}$-term* is an expression constructed, as described below, from the operators in $\mathcal{V}_{ops}$ and some (infinite) set of variables over $\mathcal{V}$. Associated with each term is a sort known as the sort of that term.

- Any variable v over $\mathcal{V}$ is a $\mathcal{V}$-term. Its sort is $v.sort$.

- For any operator op in $\mathcal{V}_{ops}$ with signature $T_1, \ldots, T_n \to T$ and for any terms $t_1, \ldots, t_n$ of sorts $T_1, \ldots, T_n$, the expression $op(t_1, \ldots, t_n)$ is a $\mathcal{V}$-term. Its sort is the range sort of op.

- For any $\mathcal{V}$-term t of sort `Bool` and any variable v over $\mathcal{V}$, the expressions $\forall v\ t$ and $\exists v\ t$ are $\mathcal{V}$-terms. Their sort is `Bool`. (The symbols $\forall$ and $\exists$, as well as the expressions $\forall v$ and $\exists v$, are called *quantifiers*. The term t is the *scope* of the quantifiers $\forall v$ and $\exists v$.)

An occurrence of a variable in a term is *bound* if it occurs within the scope of a quantifier over that variable. An occurrence of a variable in a term is *free* if it is not bound.

For any term t, any variable v, and any term s with no free variables, $t[v \leftarrow s]$ is the term obtained from t by replacing each free occurrence of v by s.

A *formula* is a term of sort `Bool`. A *sentence* is a formula with no free variables.

11.2 Semantics

Given a precise syntax for expressions in multisorted first-order logic, we now provide a precise semantics. Readers may wish to skim this section, which essentially defines expressions to mean what they seem to mean. The point here is that "meaning" has meaning only with respect to particular mathematical objects, called structures. For example, an expression $x \cdot y$ might denote the product of two numbers, the composition of two functions, or the concatentation of two strings, and a statement such as $\forall x \forall y (x < y \Rightarrow \exists z (x < z \wedge z < y))$ might be true about some structures (e.g., the rational or real numbers), but false about others (e.g., the integers).

For any vocabulary $\mathcal{V}$, a *$\mathcal{V}$-structure* $\mathcal{S}$ is a map $[\![\cdots]\!]_{\mathcal{S}}$ with domain $\mathcal{V}$ such that

- for each sort T in $\mathcal{V}$, $[\![T]\!]_{\mathcal{S}}$ is a nonempty set (called the *carrier of* T) that is disjoint from $[\![T']\!]_{\mathcal{S}}$ for any other sort T' in $\mathcal{V}$, and

- for each operator symbol op with signature $T_1, \ldots, T_n \to T$ in $\mathcal{V}$, $[\![op]\!]_{\mathcal{S}}$ is a total function from $[\![T_1]\!]_{\mathcal{S}} \times \cdots \times [\![T_n]\!]_{\mathcal{S}}$ to $[\![T]\!]_{\mathcal{S}}$.

When a vocabulary $\mathcal{V}$ contains the symbols `Bool`, `true`, `false`, $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $=$, or $\neq$, as described in Section 11.1, we restrict our attention to $\mathcal{V}$-structures that interpret these symbols as in Figure 11.1.

For any vocabulary $\mathcal{V}$, any $\mathcal{V}$-structure $\mathcal{S}$, and any $\mathcal{V}$-term t with no free variables, the *denotation* $[\![t]\!]_{\mathcal{S}}$ of t is defined recursively, as follows:

- $[\![op(t_1, \ldots, t_n)]\!]_{\mathcal{S}} = [\![op]\!]_{\mathcal{S}}([\![t_1]\!]_{\mathcal{S}}, \ldots, [\![t_n]\!]_{\mathcal{S}})$

- $[\![\exists v\ t']\!]_{\mathcal{S}} = true$ iff $[\![t'[v \leftarrow c_v]]\!]_{\mathcal{S}'} = true$ for some $(\mathcal{V} \cup \{c_v\})$-structure $\mathcal{S}'$ that agrees with $\mathcal{S}$ on $\mathcal{V}$ (i.e., such that $[\![v]\!]_{\mathcal{S}} = [\![v]\!]_{\mathcal{S}'}$ for all $v \in \mathcal{V}$), where c_v is a constant symbol not in $\mathcal{V}_{ops}$ that has sort $v.sort$.

- $[\![\forall v\ t']\!]_{\mathcal{S}} = true$ iff $[\![t'[v \leftarrow c_v]]\!]_{\mathcal{S}'} = true$ for all $(\mathcal{V} \cup \{c_v\})$-structures $\mathcal{S}'$ that agree with $\mathcal{S}$ on $\mathcal{V}$, where c_v is a constant symbol not in $\mathcal{V}_{ops}$ that has sort $v.sort$.

$[\![\texttt{Bool}]\!]_{\mathcal{S}} = \{true, false\}$
$[\![\texttt{true}]\!]_{\mathcal{S}} = true$
$[\![\texttt{false}]\!]_{\mathcal{S}} = false$
$[\![\neg]\!]_{\mathcal{S}}(x) = true$ iff $x = false$
$[\![\wedge]\!]_{\mathcal{S}}(x, y) = true$ iff $x = true$ and $y = true$
$[\![\vee]\!]_{\mathcal{S}}(x, y) = true$ iff $x = true$ or $y = true$
$[\![\Rightarrow]\!]_{\mathcal{S}}(x, y) = true$ iff $x = false$ or $y = true$
$[\![\Leftrightarrow]\!]_{\mathcal{S}}(x, y) = true$ iff $x = y$
$[\![=]\!]_{\mathcal{S}}(x, y) = true$ iff $x = y$
$[\![\neq]\!]_{\mathcal{S}}(x, y) = true$ iff $x \neq y$
$[\![\texttt{if\_\_then\_\_else\_\_}]\!]_{\mathcal{S}}(true, x, y) = x$
$[\![\texttt{if\_\_then\_\_else\_\_}]\!]_{\mathcal{S}}(false, x, y) = y$

Figure 11.1: Standard interpretation of boolean sort and logical operators

11.3 Further terminology

In the following definitions, $\mathcal{V}$ is a vocabulary, ϕ is a $\mathcal{V}$-sentence, and T and T' are sets of $\mathcal{V}$ sentences.

$\mathcal{S}$ is a *model* of ϕ iff ϕ is true in $\mathcal{S}$, that is, iff $[\![\phi]\!]_{\mathcal{S}} = true$.

T is *consistent* iff there is a $\mathcal{V}$-structure that is a model of every sentence in T.

ϕ is a *(logical) consequence* of T iff ϕ is true in every model of T.

T is a *theory* iff it is closed under logical consequence. It is easy to see that, if T is a theory, then T is consistent iff $false \notin T$.

A theory T is an *extension* of a theory T' iff $T' \subseteq T$. It is easy to see that T is an extension of T' iff every sentence in T' is a consequence of T.

An extension T of T' is *conservative* iff T' is a set of $\mathcal{V}'$ sentences for some $\mathcal{V}' \subseteq \mathcal{V}$ and every $\mathcal{V}'$-sentence in T is also in T'. In other words, an extension T of T' is conservative iff the vocabulary of T includes that of T', but all consequences of T in the vocabulary of T' are already consequences of T'.

When $\mathcal{S}$ is clear from the context, we write $[\![\cdots]\!]$ for $[\![\cdots]\!]_{\mathcal{S}}$.

12 Lexical syntax

We describe the syntax of IOA (and LSL) using a context-free grammer expressed in an extended version of BNF. Uppercase words and symbols enclosed in single quotation marks are terminal symbols in the grammar. All other words are nonterminal symbols. If x and y are grammatical units, then the following notations have the indicated meanings.

Notation	Meaning
$x\ y$	an x followed by a y
$x \mid y$	an x or a y
$x?$	an optional x
$x*$	zero or more x's
$x+$	one or more x's
$x, *$ or $x; *$	zero or more x's, separated by commas or semicolons
$x, +$ or $x; +$	one or more x's, separated by commas or semicolons

The lexical grammar of IOA uses the following symbols:

- Punctuation marks: `,  :  ;  (  )  {  }  [  ]  __  :=`
- Reserved words: **assumes**, **automaton**, **axioms**, **backward**, **by**, **case**, **choose**, **components**, **const**, **do**, **eff**, **else**, **elseif**, **enumeration**, **ensuring**, **fi**, **for**, **forward**, **from**, **hidden**, **if**, **in**, **initially**, **input**, **internal**, **invariant**, **local**, **od**, **of**, **output**, **pre**, **signature**, **simulation**, **states**, **tasks**, **then**, **to**, **transitions**, **tuple**, **type**, **union**, **where**.
- Beginning comment character: `%`
- `IDENTIFIER`s for variables, types, and functions: sequences of letters, digits, apostrophes, and underscores (except that two underscores cannot occur consecutively). The LaTeX identifiers for the Greek letters can also be used as identifiers, as can the two strings `\bot` and `\top`.
- `OPERATOR`s: sequences of the characters `- ! # $ & * + . < = > ? @ ^ | ~ /` or a backslash (`\`) followed by one of these characters, by one of the characters `_ \ %`, or by an identifier (other than a name of a Greek letter, `\bot`, or `\top`).
- Whitespace: space, tab, newline.
- Reserved for future use: `` ` " ``

13 IOA specifications

An IOA specification contains four different kinds of units.

- Type definitions, used to associate sets of values and operators with state components and parameters (see Section 14).
- Automaton definitions (see Sections 16 and 17).
- Assertions about automata, e.g., invariant and simulation relations (see Section 18).
- Axiomatizations of abstract data types, formalized in the Larch Shared Language (LSL), which provide the syntax and semantics for types and operators appearing in the other three kinds of units (see Part IV).

Syntax of IOA specifications

```
specification ::= trait | ioaSpec
ioaSpec       ::= (axioms | typeDef | automatonDef | assertion)+
```

The IOA tools described in Appendix B expect the specification for a trait `T` to be stored in a file named `T.lsl`. They expect an `ioaSpec` to be stored in a file with a name of the form `<filename>.ioa`.

The syntax and semantics for the constructs mentioned, but not defined, here can be found in Section 20 (`trait`), Section 14 (`axioms`, `typeDef`), Section 16 (`automatonDef`), and Section 18 (`assertion`).

14 Type and type constructor definitions

The syntax and semantics of the types that appear in automaton definitions are given by built-in or user-supplied LSL traits (see Sections 9 and 20). A type can be a primitive type, named by a single identifier, or a compound type, named by applying a type constructor to a list of simpler types. Each `type` denotes a `sort` (see Section 21), namely, the `sort` that is lexically identical to the `type`.

Syntax of type declarations

```
typeDef         ::=  'type' type '=' shorthand
axioms          ::=  'axioms' axiomSet,+
axiomSet        ::=  traitRef
                  |  traitId 'for' typeConstructor '[' '__',* ']'
type            ::=  simpleType | compoundType
simpleType      ::=  IDENTIFIER
compoundType    ::=  typeConstructor '[' type,+ ']'
typeConstructor ::=  IDENTIFIER
```

The syntax and semantics for `shorthand` and `traitRef` can be found in Section 9.8 and 24.

The *vocabulary*, $\mathcal{V}_{spec}$, of an `ioaSpec` is the union of the vocabularies of its `typeDefs`, its `axioms`, and the traits defining the built-in primitive types (see Appendix A). The vocabulary of a `typeDef` contains the defined `type` and the vocabulary of the defining `shorthand` (see Section 23). The *vocabulary* of an **axioms** is the union of the vocabularies of its `traitRefs` (see Section 24). (The traits named by the `traitIds` associated with `typeConstructors` do not contribute to this vocabulary.)

The *type constructors* associated with an `ioaSpec` are the built-in type constructors (see Section 8) and the `typeConstructors` defined in the `axiomSets` in the `ioaSpec`.

The *global theory*, $\mathcal{T}_{spec}$, of an `ioaSpec` is the union of the theories of its `typeDefs` and its `axioms`. The theory of a `typeDef` is the theory of its `shorthand` (see Section 23). The *theory* of an `axioms` is the union of the theories of its `traitRefs`. (The traits named by the `traitIds` associated with `typeConstructors` do not contribute to $\mathcal{T}_{spec}$.)

Static semantics

- A `type` can be defined in at most one `shorthand` in an `ioaSpec`. *Editorial note: Determine whether the front-end tool checks this. What about a definition inside a trait?*

- A `typeConstructor` can be defined in at most one `axiomSet` in an `ioaSpec`.

- The arity of a `typeConstructor` defined in an `axiomSet` is the number of `__` placeholders between the brackets following the `typeConstructor`. The trait named by a `traitId` in an `axiomSet` must have the same number of `traitFormals` as the arity of the `typeConstructor`; each of those `traitFormals` must name a sort (not an operator) in the referenced trait.

Logical semantics

- $\mathcal{T}_{spec}$ must be consistent.

15 Automaton definitions

An automaton can be a primitive automaton or a composition of other automata. Its definition can be parameterized by a list of types and/or constants.

Syntax of automaton definitions

```
automatonDef      ::= 'automaton' automatonName automatonFormals?
                      assumptions? (simpleBody | composition)
automatonName     ::= IDENTIFIER
automatonFormals ::= '(' automatonFormal,+ ')'
automatonFormal   ::= IDENTIFIER,+ ':' (type | 'type')
assumptions       ::= 'assumes' traitRef,+
```

The syntax and semantics for the constructs mentioned, but not defined, here can be found in Section 16 (`simpleBody`), Section 17 (`composition`), Section 14 (`type`), and Section 24 (`traitRef`).

An `automatonFormal` that contains the keyword **type** denotes a sequence of *formal types*, each element of which is simple sort (cf. Section 21) corresponding to an `IDENTIFIER` in the `automatonFormal`. An `automatonFormal` that contains a `type` denotes a sequence of *formal parameters*, each element of which is a constant of the sort associated with the `type`. An `automatonFormals` denotes the sequence of *automaton formals* obtained by concatenating the sequences of formal types and formal parameters in its `automatonFormals`.

The *vocabulary*, $\mathcal{V}_A$, of an `automatonDef` for an automaton with `automatonName` A in an `ioaSpec` is the union of $\mathcal{V}_{spec}$ with the vocabularies of the `traitRefs` in its `assumptions`, enriched by the automaton formals of the `automatonDef`. The *local theory*, $\mathcal{T}_A$, of this `automatonDef` is the union of $\mathcal{T}_{spec}$ and the theories of the `traitRDefs` in the `assumptions` of the `automatonDef`.

The *closures*, $cl(\mathcal{V}_A)$ and $cl(\mathcal{T}_A)$, of $\mathcal{V}_A$ and $\mathcal{T}_A$ are the smallest sets X and Y such that

- $\mathcal{V}_A \subseteq X$,

- $\mathcal{T}_A \subseteq Y$, and

- for any sorts $S_1, \ldots, S_n$ in X and any n-ary type constructor T associated with a trait Tr in the `ioaSpec` in which A is defined, the vocabulary of the `traitRef` $Tr(S_1, \ldots, S_n)$ is a subset of X and the theory of this `traitRef` is a subset of Y.

Static semantics

- There can be at most one automatonDef for a given automatonName in an ioaSpec.
- The automaton formals in an automatonDef must be distinct.
- The sort associated with a formal type in an automatonDef must not be in $\mathcal{V}_{spec}$.
- The sort of each formal parameter in an automatonDef for A must be in $cl(\mathcal{V}_A)$.

Logical semantics

- $cl(\mathcal{T}_A)$ must be consistent for any automatonDef for A.

16 Primitive automata

16.1 Primitive automaton definitions

A primitive automaton is defined by its action signature, its states, its transitions, and (optionally) a partition of its actions into tasks.

Syntax of primitive automaton definitions

```
simpleBody       ::= 'signature' formalActionList+ states transitions tasks?
formalActionList ::= actionType formalAction,+
actionType       ::= 'input' | 'output' | 'internal'
formalAction     ::= actionName (actionFormals where?)?
actionName       ::= IDENTIFIER
actionFormals    ::= '(' actionFormal,+ ')'
actionFormal     ::= IDENTIFIER,+ ':' type | 'const' term
where            ::= 'where' predicate
```

The syntax and semantics of states, transitions, and tasks are given in Sections 16.2, 16.3 and 16.4, respectively. The syntax and semantics of terms and predicates are given in Section 22.

Each actionFormal denotes a sequence of terms. If the actionFormal contains the keyword **const**, this sequence contains the single term following the keyword. Otherwise, this sequence contains a *formal parameter* (i.e., a constant) of the sort associated with the type in the actionFormal for each IDENTIFIER in the actionFormal.

The *action pattern* of a formalAction consists of its actionName, the sequence of sorts of its actionFormals, and its actionType (**input**, **output**, or **internal**).

Static semantics

- An actionName can appear in at most one action pattern with each of the three actionTypes in a simpleBody.
- An actionName must be associated with the same sequence of sorts in each action pattern in which it appears.
- Each formal parameter must be distinct from any other formal parameter of the same type in the same actionFormals, as well as from any automatonFormal.

- The type of each `actionFormal` must be in $cl(\mathcal{V}_A)$.

- Each constant or operator in a `term` following the keyword **const** in an `actionFormal`, or in a `predicate` in a `where` clause, must be an `actionFormal` in that action or in $cl(\mathcal{V}_A)$. Each variable in such a `term` or `predicate` must be a bound variable with a sort in $cl(\mathcal{V}_A)$.

- The type of a `term` used as a **const** `actionFormal` cannot be **type**.

Logical semantics

- *Editorial note: Insert formal definition of the action signature of an automaton.*

- A `formalAction` of the form `name(x: S,` **const** `t)`, where the term `t` has type `T`, is equivalent to the `formalAction` `name(x: S, y: T)` **where** `y = t`, where `y: T` is a new formal parameter.

16.2 Automaton states

States are records of state variables. An initial value for each variable can be specified by an expression; instead, or in addition, the initial values of all state variables can be restricted by a predicate. Expressions and predicates are terms.

Syntax of state variable definitions

```
states ::= 'states' state,+ ('initially' predicate)?
state  ::= IDENTIFIER ':' type (':=' value)?
value  ::= term | choice
choice ::= 'choose' (variable 'where' predicate)?
```

The syntax and semantics of `predicate`, `term`, and `variable` are given in Section 22.
Each `state` defines a *state variable*, which is a constant with the given `IDENTIFIER` and `type`.

Static semantics

- Each state variable must be distinct from all other state variables and from all formal parameters of the automaton.

- The type of each state variable must be in $cl(\mathcal{V}_A)$.

- The type of the initial `value` assigned to a state variable must be the same as the type of that state variable.

- Each constant or operator in a `term` assigned as the initial `value` of a state variable must be in $cl(\mathcal{V}_A)$. Each variable in this `term` must be a bound variable with a sort in $cl(\mathcal{V}_A)$.

- Each constant, operator, or variable in the `predicate` in a `choice` is similarly limited, except that the variable following the keyword **choose** can also appear in the `predicate`. The type of this variable, if specified, must be the same as the type of the state variable. The identifier for this variable must be distinct from that of any parameter or state variable of the automaton that has the same sort.

- Each constant, operator, or variable in the `predicate` restricting the initial values of the state variables is similarly limited, except that state variables can also appear in the `predicate`.

Logical semantics

- *Editorial note: Insert formal definition of the sets of states and start states of an automaton. Consider phrasing in terms of "For any model of $\mathcal{T}_A$, ...*
- The set of start states, determined by the assignments and/or allowed by the predicates, must be nonempty. *Editorial note: Rephrase in terms of formal definition.*

16.3 Automaton transitions

Transitions are specified using precondition/effect notation. Preconditions are boolean-valued predicates. Effects can be described in terms of simple programs and/or restricted by predicates relating the poststate to the prestate.

Syntax of transition relations

```
transitions     ::=   'transitions' transition+
transition      ::=   actionHead precondition? effect?
actionHead      ::=   actionType actionName
                      (actionActuals where? transitionCase)?
actionActuals   ::=   '(' term,+ ')'
actionActuals   ::=   '(' term,+ (';' localParameters)? ')'
                    | '(' localParameters ')'
localParameters ::=   'local' variableList,+
variableList    ::=   IDENTIFIER,+ ':' type
precondition    ::=   'pre' predicates
predicate       ::=   predicate;+ ';'?
effect          ::=   'eff' program ('ensuring' predicate)?
program         ::=   statement;+
statement       ::=   assignment | conditional | loop
assignment      ::=   lvalue ':=' value
lvalue          ::=   variable
                    | lvalue '[' term,+ ']'
                    | lvalue '.' IDENTIFIER
conditional    ::=   'if' predicate 'then' program
                     ('elseif' predicate 'then' program)*
                     ('else' program)? 'fi'
loop           ::=   'for' IDENTIFIER qualification
                     ('in' | 'where') term 'do' program 'od'
```

The syntax and semantics of `predicate`, `qualification`, `variable`, and `term` are given in Section 22.

Each `localParameters` defines a sequence of *local parameters*, which are constants with the `IDENTIFIER`s and `type`s specified in its `variableList`s. *Editorial note: Replace* `variableList` *by* `constantList`.

Static semantics

- Transitions must be specified for all `actionName`s in the signature of the automaton.

- The `actionNames` for which `transitions` are specified must be in the signature of the automaton.
- The `actionActuals` for each transition must match, both in number and in type, the `actionFormals` for the `actionName`.
- The types of variables appearing in `actionActuals` must be determined uniquely by the types of the `actionActuals`. These variables are declared implicitly by their occurrence in the `actionActuals` and have no relation to variables used as `actionFormals`.
- No `precondition` is allowed for an input action.
- Each local parameter must be distinct from all other local parameters, from all formal parameters of the automaton, from all state variables, and from all variables in the `actionActuals` of the action.
- All operators, constants, and variables in a `predicate` in a `precondition` or `conditional`, or in a `lvalue` or `value` in an `assignment`, must be
  - in $cl(\mathcal{V}_A)$,
  - variables introduced in the `actionActuals`,
  - local parameters of the action,
  - state variables of the automaton,
  - variables introduced in a `loop` containing the `predicate` or `term`, or
  - variables in the scope of a `quantifier` in the `predicate` or `term`.
- All constants, operators, and variables in the `predicate` in an **ensuring** clause must satisfy the same restrictions or be primed versions of state or **local** variables that are modified by some `assignment` in the `program` in the `effect` clause. For example, if `queue` is a state variable that appears on the left side of an assignment, then both `queue` and `queue`$'$ are allowed in the predicate.
- The type of the variable in a loop (i.e., the type associated with the `qualification`) must be in $cl(\mathcal{V}_A)$. The scope of the variable is the loop. The variable need not be distinct from other constants and variables used outside that scope.

Logical semantics

- The `where` clause in each transition definition is implicitly conjoined with the `where` clause for the corresponding entry in the signature.
- Each `transition` defines a binary relation between states of the automaton. This relation is defined by the formula

  $$\exists h \ldots (pre(s) \wedge \mathit{eff}(s, s') \wedge ensuring(s, s'))$$

  where
  - $h \ldots$ are the local parameters, if any, in the transition,
  - $pre(s)$ is the `predicate` in the `precondition`,

- $\mathit{eff}(s, s')$ is a formula obtained by translating the `program`, if any, in the `effect`, as described below, and
- $\mathit{ensuring}(s, s')$ is the `predicate`, if any, in the **ensuring** clause in the `effect`.

- The semantics of a `program` P is defined by translating it into an **ensuring** clause eff_P, as indicated in the following table. In that table, s and s' represent states, v is a state variable (with value $s.v$ in state s), w is an arbitrary state variable distinct from v, t is a term, p is a predicate, and P_1 and P_2 are programs.

program P	eff_P
$v := t$	$s'.v = t \wedge s'.w = s.w$
P_1 ; P_2	$\exists s''(\mathit{eff}_{P_1}(s, s'') \wedge \mathit{eff}_{P_2}(s'', s'))$
if p **then** P_1 **fi**	$(p \to \mathit{eff}_{P_1}(s, s')) \wedge (\neg p \to s' = s)$
if p **then** P_1 **else** P_2 **fi**	$(p \to \mathit{eff}_{P_1}(s, s')) \wedge (\neg p \to \mathit{eff}_{P_2}(s, s'))$
for v **in** t **do** P_1 **od**	$\forall x(x \in t \Rightarrow \mathit{eff}_{v:=x;P_1}(s, s'))$

- The formula $\mathit{eff}(s, s')$ obtained by translating a `program` in an `effect` must be consistent.
- Identifiers for state variables in **ensuring** clauses refer to the values of the variables in the prestate, i.e., in the state before the transition is executed. Primed versions of these identifiers refer to the values of the variables in the poststate, i.e., in the state after the transition is executed.

Note that:

- Statements in a `program` are executed sequentially, not in parallel as in UNITY [2].
- State variables that do not appear on the left side of an `assignment` in a branch through the `program` in an `effect` clause are assumed to be unchanged on that branch.

16.4 Automaton tasks

Tasks define a partition of the actions of an automaton.

Syntax of tasks

```
tasks     ::= 'tasks' task;+
task      ::= '{' actionSet '}' forClause?
actionSet ::= actionName ('(' term,+ ')' where?)?
forClause ::= 'for' variableList,+ where?
```

Static semantics

- Each `actionName` in a task must be an internal or output action of the automaton.
- The number of `actionActuals` for an `actionName` must equal the number of `actionFormals` in the automaton's signature for that `actionName`.
- The type of each `actionActual` must be the same as that of the corresponding `actionFormal`.

- All operators, constants, and identifiers in a `term` in an `actionActual` or in a `where` clause must be in $cl(\mathcal{V}_A)$ or defined exactly once in a `for` clause associated with the task. *Editorial note: check this.*

Logical semantics

- The task definitions must define a partition of the set of all non-input actions.
- If no `tasks` is present, then all non-input actions are treated as belonging to a single task.

17 Composite automata

Automata can be constructed by composing a set of component automata. Composite automata identify actions in different component automata that have the same name and the same values for their parameters; when any component automaton performs a step involving an action π, so do all component automata that have π in their signatures. The hiding operator reclassifies output actions as internal actions.

Syntax of composite automata definitions

```
composition       ::=  'components' component;+ ('hidden' actionSet)?
component         ::= componentTag (':' componentDef)? where?
componentTag      ::= componentName componentFormals?
componentName     ::= IDENTIFIER
componentFormals ::= '[' variableList,+ ']'
componentDef      ::= automatonName automatonActuals?
automatonActuals ::= '(' (term | type),+ ')'
```

Static semantics

- If a `component` does not contain a `componentDef`, it is assumed to have one in which the `automatonName` is the same as the `componentName` and the `automatonActuals` are the `variables` (considered as `terms`) in the `componentFormals`.
- The variables used as `componentFormals` must be distinct from each other and from any `automatonFormal`.
- The type of each `componentFormal` must be in $cl(\mathcal{V}_A)$.
- Each `automatonName` must be defined in an `automatonDef`.
- The numbers and types of the `automatonActuals` must match those of the corresponding `automatonFormals`.
- All constants, operators, and variables in `terms` used as `automatonActuals` must be in $cl(\mathcal{V}_A)$, bound variables, or `componentFormals`.
- Similarly named actions in different component automata must have the same number and types of parameters.

- The set of internal actions for each component must be disjoint from the set of all actions for each of the other components.
- The set of output actions for each component must be disjoint from the set of output actions for each of the other components.
- Each `actionName` in an `actionSet` must occur as the name of an output action in the signature of at least one of the component automata.

Logical semantics

- Each action of the composition must be an action of only finitely many component automata.
- The signature of the composition is the union of the signatures of the component automata.
- An action is an output action of the composition if it is an output action of some component automaton.
- An action is an input action of the composition if it is an input action of some component automaton, but not an output action of any component.
- An action is an internal action of the composition if it is an internal action of some component automaton.
- The set of states of the composition is the Cartesian product of the sets of states of the component automata.
- The set of start states of the composition is the Cartesian product of the sets of start states of the component automata.
- A triple (s, π, s') is in the transition relation for the composite automaton if, for every component automaton C, (s_C, π, s'_C) is a transition of C when π is an action of C and $s_C = s'_C$ when π is not an action of C.

Editorial note: This document needs to describe the notations that can be used for state variables of composite automata in invariants and simulation relations. A description of these notations can be found in [14].

Editorial note: State that one can prove a theorem that allows replacement of one component by another that implements it without affecting the traces of the composite automaton.

18 Statements about automata

Assertions about automata make claims about invariants preserved by the actions of the automata or about simulation relations between two automata.

Syntax of invariant and simulation relations

```
assertion  ::= invariant | simulation
invariant  ::= 'invariant' 'of' automatonName ':' predicate
simulation ::= ('forward' | 'backward') 'simulation' 'from'
               automatonName 'to' automatonName ':' predicate
```

Static semantics

- Each `automatonName` must have been defined previously in an `automatonDef`.
- All operators, constants, and variables in a `predicate` in an `assertion` must be
  - in $cl(\mathcal{V}_A)$ for (one of) the named automata,
  - state variables of (one of) the named automata, or
  - variables in the scope of a `quantifier` in the `predicate`.

Logical semantics

- An invariant must be true in all reachable states of the automaton.
- The proof obligations for simulation relationships are as defined in Section 1.4.

Part IV
LSL Reference Manual

An LSL specification defines a theory in multisorted first-order logic. It presents a set of axioms for that theory. It may also present claims about the intended consequences of these axioms.

19 Lexical syntax

The lexical grammar of LSL is the same as that of IOA (Section 12), except that it uses the following list of reserved words: **asserts**, **assumes**, **by**, **converts**, **else**, kenumeration, **exempting**, **for**, **freely**, **generated**, **if**, **implies**, **includes**, **introduces**, **of**, **partitioned**, **sort**, **then**, **trait**, **traits**, **tuple**, **type**, **union**, **with**.

20 Traits

The basic unit of specification in LSL is a *trait*, which defines a set of axioms for a logical theory and which makes claims about the consequences of that theory. The header for a trait specifies its name and an optional list of formal parameters, which can be used in references to other traits (see Section 24). The body of the trait consists of optional references to subtraits (Section 24) intermixed with shorthands defining sorts (Section 23), optionally followed by sort and operator declarations (Section 21), axioms (Section 22), and claimed consequences of the axioms (Section 25).

Syntax of traits

```
trait     ::= traitId traitFormals? ':' 'trait' traitBody
traitId   ::= IDENTIFIER
traitBody ::= (subtrait | sort shorthand)* opDcls? axioms? consequences?
```

21 Sort and operator declarations

Sorts in LSL can be simple sorts, which are named by a single identifier, or compound sorts, which are named by a sort constructor applied to a list of simpler sorts. Table 21.1 summarizes the ways in which operators can be declared and used in several different kinds of notations for terms.

Placeholders in operator declarations indicate where the operators arguments are placed. Signatures in operator declarations indicate the sorts of the arguments for an operator (its *domain* sorts) and the sort of its value (its *range* sort).

Syntax of operator declarations

```
opDcls        ::=  'introduces' opDcl+
opDcl         ::=  name,+ ':' signature ','?
name          ::=  'if' '__' 'then' '__' 'else' '__'
                 | '__'? OPERATOR '__'?
                 | '__'? openSym '__',* closeSym '__'?
                 | '__'? '.' IDENTIFIER
                 | IDENTIFIER
```

Sample declaration	Form of term	Sample use
`f: Int -> Int`	functional	$f(i)$
`min: Int, Int -> Int`	”	$min(i,j)$
`0: -> Int`	zeroary	0
`__<__: Int, Int -> Bool`	infix	$i < j$
`-__: Int, Int -> Int`	prefix	$-i$
`__!: Int, Int -> Int`	postfix	$i!$
`__.last: Seq[Int] -> Int`	”	$s.last$
`__[__]: A, Int -> V`	bracketed	$a[i]$
`{__}: E -> Set[E]`	”	$\{x\}$
`{}: -> Set[E]`	”	$\{\}$
`if__then__else__: Bool, S, S -> S`	conditional	**if** $x < 0$ **then** $-x$ **else** x
	quantified	$\forall x \exists y (x < y)$

Table 21.1: Sample operator declarations and use in terms

```
openSym         ::=   '[' | '{' | '\(' | '\<'
                    | '\langle' | '\lfloor' | '\lceil'
closeSym        ::=   ']' | '}' | '\)' | '\>'
                    | '\rangle' | '\rfloor' | '\rceil'
operator        ::=   name (':' signature)?
signature       ::=   domain '->' range
domain          ::=   sort,*
range           ::=   sort
sort            ::=   simpleSort | compoundSort
simpleSort      ::=   IDENTIFIER
compoundSort    ::=   sortConstructor '[' sort,+ ']'
sortConstructor ::=   IDENTIFIER
```

The optional comma at the end of an `opDcl` is required if the following `opDcl` begins with a left bracket.

Static semantics

- The number of `__` placeholders in the `name` in an `opDcl` must be the same as the number of sorts in the domain of its `signature`.
- The `__` placeholder cannot be omitted from a `name` of the form `__.IDENTIFIER` in an `opDcl`.
- The signature of the operators **`true`** and **`false`** must be →`Bool`. Declarations for these operators are built into LSL.

- The signature of the logical operators $\Leftrightarrow$, $\Rightarrow$, $\wedge$, and $\vee$ must be `Bool,Bool→Bool`. Declarations for these operators are built into LSL.

- The signature of the operators $=$ and $\neq$ must be `S,S→Bool` for some sort `S`. Declarations for these operators are built into LSL for each sort `S` that occurs in an `opDcl` or `shorthand` (see Section 23).

- The signature of the operator `if__then__else__` must be `Bool,S,S→S` for some sort `S`. A declaration for this operator is built into LSL for each sort `S` that occurs in an `opDcl` or `shorthand` (see Section 23).

Logical semantics

- A **sort** denotes a non-empty set of elements.[14]

- Different sorts denote disjoint sets of elements.

- An `opDcl` defines a list of *operators*, each with a given `name` and `signature`.

- Each operator denotes a total function from tuples of elements in its domain sorts to an element in its range sort.

Formal semantics

22 Axioms

Axioms in LSL are either formulas in multisorted first-order logic or abbreviations for sets of formulas. A limited amount of operator precedence, as illustrated in the following table, is used when parsing terms (see Section 9.1).

Unparenthesized term	Interpretation
$x - y - z$	$(x - y) - z$
$a = b + c \Rightarrow b < s(a)$	$(a = (b + c)) \Rightarrow (b < s(a))$
$a.b.c!$	$((a.b).c)!$
$\neg p \wedge \neg x.pre$	$(\neg p) \wedge (\neg(x.pre))$
$\exists x (x < c) \Rightarrow c > 0$	$(\exists x (x < c)) \Rightarrow (c > 0)$
$\forall x \exists y\ x < y$	$(\forall x \exists y\ x) < y$
$a < b + c$	Error
$p \wedge q \vee r$	Error
$p \Rightarrow q \Rightarrow r$	Error

[14]LSL accords syntactic, but not semantic, meaning to compound sorts.

Syntax of axioms

```
axioms        ::=   'asserts' varDcls? axiom;+ ';'?
varDcls       ::=   'with' (IDENTIFIER,+ qualification)+
qualification ::=   ':' sort
axiom         ::=   predicate
                  | 'sort' sort ('generated' 'freely'? | 'partitioned')
                    'by' operator,+
predicate     ::=   term
term          ::=   IF term THEN term ELSE term
                  | subterm
subterm       ::=   subterm (OPERATOR subterm)+
                  | (quantifier | OPERATOR)* OPERATOR secondary
                  | (quantifier | OPERATOR)* quantifier primary
                  | secondary OPERATOR*
quantifier    ::=   ('\A' | '\E') variable
variable      ::=   IDENTIFIER qualification?
secondary     ::=   primary
                  | primary? bracketed ('.'? primary)?
primary       ::=   primaryHead (qualification | '.' primaryHead)*
primaryHead   ::=   IDENTIFIER ('(' term,+ ')')?
                  | '(' term ')'
bracketed     ::=   openSym term,* closeSym qualification?
```

Static semantics

- Each operator in an axiom must be a built-in operator, declared in an operator declaration (Section 21), introduced by a `shorthand` for a sort (Section 23), or declared in a subtrait (Section 24).
- Each sort in a `qualification` must have been declared.
- No variable may be declared more than once in a `varDcls`.
- A variable cannot be declared to have the same identifier and sort as a constant (i.e., as a zero-ary operator).
- There must be unique assignment of declared operators and variables to the identifiers, `OPERATOR`s, `openSym`s, and `closeSym`s in a term so that the arguments of each declared operator have the appropriate sorts and so that every qualified subterm has the appropriate sort.
- If a `subterm` contains two `OPERATOR`s, those `OPERATOR`s must either be the same or have different parsing precedences.
- The sort of a predicate must be `Bool`.
- The sort named in a **generated by** or a **partitioned by** must have been declared.
- The range of each operator in a **generated by** must be the named sort.

- At least one of the operators in a **generated by** must not have the named sort in its domain.
- Each operator in a **partitioned by** must have the named sort in its domain.
- The list of operators in a **generated by** or **partitioned by** must not contain duplicates.

Logical semantics

- Let S be a sort, $f_1, \ldots, f_n$ be operators with range S, and $\vec{v_1}, \ldots, \vec{v_m}$ be disjoint sequences of variables such that $f_i(\vec{v_i})$ is a well-formed term for $1 \leq i \leq n$. We call f_i a *basis operator* if no variable in $\vec{v_i}$ has sort S; otherwise we call f_i an *inductive operator*. Then the axiom **sort** S **generated by** $f_1, \ldots, f_n$ is equivalent to the infinite set of formulas that contains, for each formula $\phi(x)$ with a free variable x of sort S, the formulas
  - $\phi(f_i(\vec{v_i}))$, for each i such that $1 \leq i \leq n$ and f_i is a basis operator, and
  - $\phi(x_1) \wedge \ldots \wedge \phi(x_k) \Rightarrow \phi(f_i(\vec{v_i}))$, for each $i, x_1, \ldots, x_k$ such that $1 \leq i \leq n$, f_i is an inductive operator, and $x_1, \ldots, x_n$ are the variables of sort S in $\vec{v_i}$.

  The axiom **sort** S **generated freely by** $f_1, \ldots, f_n$ is equivalent to the above set of formulas extended by the formulas
  - $f_i(\vec{v_i}) \neq f_j(\vec{v_j})$, for $1 \leq i < j \leq n$, and
  - $f_i(\vec{v_i}) = f_i(\vec{w}) \Rightarrow \vec{v_i} = \vec{w}$ for $1 \leq i \leq n$, where $\vec{w}$ is a sequence of variables not in $\vec{v_i}$ such that $f_i(\vec{w})$ is a well-formed term.
- Let S be a sort, x_1 amd x_2 be variables of sort S, $f_1, \ldots, f_n$ be operators, and $\vec{v_1}, \ldots, \vec{v_m}$ be sequences of variables not containing either x_1 or x_2 such that $f_i(x, \vec{v_i})$ is a well-formed term for $1 \leq i \leq n$. The axiom **sort** S **partitioned by** $f_1, \ldots, f_n$ is equivalent to the axiom

  $$\forall \vec{v_1}(f_1(x_1, \vec{v_1}) = f_1(x_2, \vec{v_1})) \wedge \ldots \wedge \forall \vec{v_n}(f_n(x_1, \vec{v_n}) = f_1(x_2, \vec{v_n})) \Rightarrow x_1 = x_2$$

  Editorial note: Generalize for functions that have multiple arguments of sort S.

23 Shorthands for sorts

LSL shorthands provide a convenient way of declaring sorts representing enumerations, tuples, and unions.

Syntax of shorthands

```
shorthand ::=  'enumeration' 'of' IDENTIFIER,+
            | ('tuple' | 'union') 'of' (IDENTIFIER,+ ':' sort),+
```

Static semantics

- The list of identifiers in an `enumeration` must not contain duplicates.
- The list of identifiers corresponding to a field of a particular sort in a `tuple` or `union` must not contain duplicates.
- Each **sort** appearing in a `shorthand` must differ from the sort of the shorthand itself.

Logical semantics

- Each shorthand is equivalent to the inclusion of a trait such as those illustrated in Figures 23.1–23.3.

```
Enumeration3: trait
  introduces
    e1, e2, e3: → S
    succ: S → S
  asserts
    sort S generated freely by e1, e2, e3;
    succ(e1) = e2;
    succ(e2) = e3
```

Figure 23.1: Expansion of the shorthand `S` **enumeration of** `e1, e2, e3`

```
Tuple3: trait
  introduces
    [__, __, __]: S1, S2, S3 → S
    __.f1: S → S1           set_f1: S, S1 → S
    __.f2: S → S2           set_f2: S, S2 → S
    __.f3: S → S3           set_f3: S, S3 → S
  asserts with x1, y1: S1, x2, y2: S2, x3, y3: S3
   sort S generated freely by [__, __, __];
   sort S partitioned by .f1, .f2, .f3;
   [x1, x2, x3].f1 = x1;    set_f1([x1, x2, x3], y1) = [y1, x2, x3];
   [x1, x2, x3].f2 = x2;    set_f2([x1, x2, x3], y2) = [x1, y2, x3];
   [x1, x2, x3].f3 = x3;    set_f3([x1, x2, x3], y3) = [x1, x2, y3]
```

Figure 23.2: Expansion of the shorthand `S` **tuple of** `f1: S1, f2: S2, f3: S3`

24 Trait references

Traits can incorporate axioms from other traits by inclusion. Traits can also contain assumptions, which must be discharged in order for their inclusion in other traits to have the intended meaning.

Syntax of trait references

```
subtrait     ::=  ('includes' | 'assumes') traitRef,+
traitRef     ::=  traitId renaming?
traitId      ::=  IDENTIFIER
renaming     ::=  '(' traitActual,+          ')'
                | '(' traitActual,* replace,+ ')'
replace      ::=  traitActual FOR traitFormal
traitActual  ::=  name | compoundSort
traitFormals ::=  '(' traitFormal,* ')'
traitFormal  ::=  name signature? | compoundSort
```

```
Union3: trait
  introduces
    __.f1: S → S1      f1: S1 → S
    __.f2: S → S2      f2: S2 → S
    __.f3: S → S3      f3: S3 → S
    tag: S → S_tag
    succ: S_tag → S_tag
    f1, f2, f3: → S_tag
  asserts with x1: S1, x2: S2, x3: S3
    sort S generated freely by f1, f2, f3;
    sort S partitioned by .f1, .f2, .f3, tag;
    f1(x1).f1 = x1;    tag(f1(x1)) = f1;
    f2(x2).f2 = x2;    tag(f2(x2)) = f2;
    f3(x3).f3 = x3;    tag(f3(x3)) = f3
```

Figure 23.3: Expansion of the shorthand `S` **union of** `f1: S1, f2: S2, f3: S3`

Static semantics

- There must not be a cycle in the assumes/includes hierarcy.
- Each `compoundSort` used as a `traitFormal` must be declared in the trait.
- Each `name` qualified by a `signature` used as a `traitFormal` must be declared as an operator in the trait.
- Placeholders can be omitted from a `name` in a `traitFormal` if there is exactly one way to supply placeholders so as to match that name with the name of a declared operator.
- Each `name` used as a `traitFormal`, but not qualified by a `signature`, must be declared as a simple sort, be declared as a sort constructor, or match the name (modulo the addition of placeholders) of exactly one declared operator.
- When a `name` used as a `traitFormal` can be interpreted in more than one way as a simple sort, sort constructor, or operator, preference is given to its intepretation first as a simple sort, second as a sort constructor, and third as an operator.
- The number of actual parameters in a trait reference must not exceed the number of formal parameters in the definition of the trait.
- No operator or sort may be renamed more than once in a renaming.
- Each `compoundSort` used as a `traitActual` must correspond to a `traitFormal` that is a sort.
- Each `name` used as a `traitActual` must be an identifier if it corresponds to a `traitFormal` that is a sort. If the `name` contains placeholders, it must correspond to a `traitFormal` that is an operator with the appropriate number of domain sorts. If the `name` contains no placeholders, there must be a unique way of adding them to match the number of domain sorts for the corresponding `traitFormal`.

Logical semantics

- The *assertions* of a trait include the axioms asserted directly in the trait, together with the (appropriately renamed) axioms asserted in all traits (transitively) included in the trait.
- The *assumptions* of a trait include the (appropriately renamed) axioms of all traits (transitively) assumed by the trait.
- When trait A includes or assumes trait B, the assertions and assumptions of A must imply the assumptions of B.
- The assertions and assumptions of any trait must be consistent.

25 Consequences

LSL traits can contain checkable redundancy in the form of consequences that are claimed to follow from their axioms.

Syntax of consequences

```
consequences ::= 'implies' varDcls? consequence;+ ';'?
consequence  ::= axiom | 'trait' traitRef,+ | conversion
conversion   ::= 'converts' operator,+ ('exempting' term,+)?
```

Static semantics

- All sorts and operators in a consequence, including those declared in an implied `traitFef`, must be declared in the implying trait.
- Each `name` in a `conversion` must correspond to exactly one declared operator (in the same manner as required for `traitFormals`).
- Each term in an `exempting` clause must contain some converted operator.

Logical semantics

- The assertions and assumptions of a trait must imply the non-conversion consequences of that trait.
- If a trait T is claimed to convert a set Ops of operators, then $op(x_1, \ldots, x_n) = op'(x_1, \ldots, x_n)$ must be a logical consequence of $T \cup T' \cup E$ for each op in Ops, where
  - op' is a new operator name,
  - T' is obtained from T by replacing each occurence of each op in Ops by op', and
  - E is the set of all formulas of the form $t = t'$, where t is an exempted term and t' is obtained from t by replacing each occurence of each op in Ops by op'.

26 Converts

Editorial note: Write this.

Part V
Appendices

A Axioms for built-in data types and type constructors

The following axiomSets (see Section 14) are included by default in an ioaSpec and refer to the traits defined in this Appendix.

```
axioms Boolean, Character, Integer, Natural, Real, String,
       Array1 for Array[__,__], Array2 for Array[__,__,__], ...,
       Mapping1 for Map[__,__], Mapping2 for Map[__,__,_], ...,
       Multiset for Mset[__], Null for Null[__],
       Sequence for Seq[__], Set for Set[__]
```

Sufficiently many axiomSets for arrays and mappings are included to define all Array and Map constructors that appear in an ioaSpec. If a traitRef (other than to Boolean) or typeConstructor in one of these axiomSets appears explicitly in an axiomSet in an ioaSpec, that axiomSet replaces the default axiomSet described below.

A.1 Auxiliary traits

The following traits defined theories that are included in various of the traits defining the built-in data types.

```
AC(T, +): trait
  % Associative-commutative operator
  introduces __+__: T, T → T
  asserts with x, y, z: T
    x + (y + z) = (x + y) + z;
    x + y = y + x

TotalOrder(T): trait
  introduces __<__, __≤__, __>__, __≥__: T, T → Bool
  asserts with x, y, z: T
    ¬(x < x);
    x < y ∧ y < z ⇒ x < z;
    x < y ∨ x = y ∨ y < x;
    x ≤ y ⇔ x < y ∨ x = y;
    x < y ⇔ y > x;
    x ≥ y ⇔ x > y ∨ x = y
  implies with x, y, z: T
    x ≤ x;
    x ≤ y ∧ y ≤ z ⇒ x ≤ z;
    x ≤ y ∨ y ≤ x;
    x < y ⇒ ¬(x = y);

DecimalLiterals(N): trait
  % This trait schema is given for documentation only.  It is
  % implicit in LSL.  Any trait that includes or assumes it
  % can use any decimal literal as a constant of sort N.
  introduces
    succ: N → N
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10: → N
    % and more, as needed by an including or assuming trait
  asserts
    1 = succ(0);
    2 = succ(1);
    3 = succ(2);
    4 = succ(3);
    5 = succ(4);
    6 = succ(5);
    7 = succ(6);
    8 = succ(7);
    9 = succ(8);
   10 = succ(9);
   %  and more, as needed by an including or assuming trait
```

A.2 Booleans

The theory for the Boolean datatype is built into LSL and cannot be overriden. It is equivalent to the theory defined by the following trait.

```
Boolean: trait
  introduces
    true, false:                          → Bool
    ¬__:                       Bool       → Bool
    __∧__, __∨__, __⇒__, __⇔__: Bool, Bool → Bool
  asserts with b: Bool
    sort Bool generated freely by true, false;
    ¬true = false;
    ¬false = true;
    (true ∧ b) = b;
    (false ∧ b) = false;
    (true ∨ b) = true;
    (false ∨ b) = b;
    (true ⇒ b) = b;
    (false ⇒ b) = true;
    (true ⇔ b) = b;
    (false ⇔ b) = ¬b
  implies with b, b1, b2, b3: Bool
    traits AC(Bool, ∧), AC(Bool, ∨), AC(Bool, ⇔)
    ¬¬b = b;
    ¬(b1 ∧ b2) ⇔ ¬b1 ∨ ¬b2;
    ¬(b1 ∨ b2) ⇔ ¬b1 ∧ ¬b2;
    b1 ∧ (b2 ∨ b3) ⇔ (b1 ∧ b2) ∨ (b1 ∧ b3);
    b1 ∨ (b2 ∧ b3) ⇔ (b1 ∨ b2) ∧ (b1 ∨ b3);
    b1 ∨ (b1 ∧ b2) ⇔ b1;
    b1 ∧ (b1 ∨ b2) ⇔ b1;
    b ∨ ¬b;
    b1 = b2 ∨ b1 = b3 ∨ b2 = b3;
    b1 ⇒ b2 ⇔ ¬b1 ∨ b2;
    (b1 ⇒ b2) ∧ (b2 ⇒ b3) ⇒ (b1 ⇒ b3)
```

A.3 Characters

```
Character: trait
  Char enumeration of
    'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
    'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z',
    'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
    'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z',
    '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
  includes TotalOrder
  asserts with a: Char
    a < succ(a)
```

In the future, IOA may provide additional notations for characters.

A.4 Integers

```
Integer: trait
  introduces
    0, 1:                                              → Int
    -__, abs, pred, succ:                     Int      → Int
    __+__, __-__, __*__, div, mod, min, max:  Int, Int → Int
    __<__, __≤__, __>__, __≥__:               Int, Int → Bool
  asserts with x, y, z: Int
   % Induction axiom
     sort Int generated by 0, succ, pred;
   % Inductive definitions
     -0 = 0;       -succ(x) = pred(-x);       -pred(x) = succ(-x);
     x + 0 = x;    x + succ(y) = succ(x+y);   x + pred(y) = pred(x+y);
     x * 0 = 0;    x * succ(y) = (x*y) + x;   x * pred(y) = (x*y) + (-x);
   % Definition of ordering
     x < succ(x);
     pred(x) < x;
   % Explicit definitions
     1 = succ(0);
     x - y = x + (-y);
     abs(x) = (if x < 0 then -x else x);
     min(x, y) = (if x < y then x else y);
     max(x, y) = (if x > y then x else y);
     x ≤ y ⇔ x < y ∨ x = y;
     x > y ⇔ y < x;
     x ≥ y ⇔ y ≤ x;
  implies with x, y, z: Int
    traits AC(Int, +), AC(Int, *), AC(Int, max), AC(Int, min);
    succ(pred(x)) = x;                       pred(succ(x)) = x;
    x + 1 = succ(x);                         x - 1 = pred(x);
    x * 1 = x;
    x + 1 ≠ x;                               x - 1 ≠ x;
    -(-x) = x;                               (-x) + x = 0;
    -(x + y) = (-x) + (-y);                  x * (-y) = -(x * y);
    pred(x) < pred(y) ⇔ x < y;               succ(x) < succ(y) ⇔ x < y;
    x < y ⇒ x < (y + 1);
    x * (y + z) = (x * y) + (x * z);
```

A.5 Natural numbers

```
Natural: trait
  includes DecimalLiterals(Nat for N)
  introduces
    pred, succ: Nat → Nat
    __+__, __-__, __*__, __**__, div, mod, min, max: Nat, Nat → Nat
    __<__, __≤__, __>__, __≥__: Nat, Nat → Bool
  asserts with x, y, z: Nat
    sort Nat generated by 0, succ;
    pred(succ(x)) = x;
    x + 0 = x;
    x + succ(y) = succ(x + y);
    x * 0 = 0;
    x * succ(y) = (x * y) + x;
    x * (y + z) = (x * y) + (x * z);
    succ(x) = succ(y) ⇔ x = y;
    x < succ(x)
  implies with x, y, z: Nat
    traits AC(Nat, +), AC(Nat, *), TotalOrder(Nat)
    0 ≤ x;
    0 < succ(x);
    ¬(x < 0);
    ¬(succ(x) ≤ x);
    succ(x) < succ(y) ⇔ x < y;
    x < succ(y) ⇒ x ≤ y
```

A.6 Real numbers

```
Real: trait
  introduces
    0, 1:                                                  → Real
    -__, abs:                                   Real       → Real
    __+__, __-__, __*__, __/__, __**__, min, max: Real, Real → Real
    Cauchy: Array[Nat, Real]       → Bool
    lim:    Array[Nat, Real], Real → Bool
  includes AC(Real, +), AC(Real, *), TotalOrder(Real), Array[Nat, Real]
  asserts with x, y, z, eps, delta: Real,
               a: Array[Nat, Real], i, j, n: Nat
    % Identity elements for arithmetic operations
      x + 0 = x;       x * 1 = x;       x / 1 = x;
    % Inverse
      x + (-x) = 0;
      x ≠ 0 ⇒ x * (y / x) = y;
    % Distributive
      x * (y + z) = (x*y) + (x*z);
    % Ordering
      x ≤ y ⇒ x + z ≤ y + z;
      0 ≤ x ∧ 0 ≤ y ⇒ 0 ≤ (x*y);
    % Completeness (every Cauchy sequences has a limit)
      Cauchy(a) ⇔
        ∀ eps (0 < eps ⇒ ∃ n ∀ i (n < i ⇒ abs(a[i]-a[i+1]) < eps));
      lim(a, x) ⇔
        ∀ eps (0 < eps ⇒ ∃ n ∀ i (n < i ⇒ abs(a[i]-x) < eps));
      Cauchy(a) ⇒ ∃ x lim(a, x);
    % Explicit definitions
      x - y = x + (-y);
      abs(x) = (if x < 0 then -x else x);
      min(x, y) = (if x < y then x else y);
      max(x, y) = (if x > y then x else y);
    % Exponentiation (define x**(i/j), extend by continuity)
      x ** 0 = 1;                          x ** 1 = x;
      x ** (y + z) = (x**y)*(x**z);        x**(y*z) = (x**y)**z;
      ∀ eps (0 < eps ⇒
        ∃ delta (0 < delta ∧
                  ∀ x ∀ y (abs(x-y) < delta ⇒ abs((z**x)-(z**y)) < eps)));
```

A.7 Strings

```
String: trait
  includes Character, Sequence(Char), TotalOrder(Seq[Char])
  asserts with a, a1, a2: Char, s, s1, s2: String
    {} < a ⊣ s;
    (a1 ⊣ s1) < (a2 ⊣ s2) ⇔ a1 < a2 ∨ (a1 = a2 ∧ s1 < s2)
```

In the future, IOA may provide additional notations for strings.

A.8 Arrays

The following traits defines one- and two-dimensional arrays. Similar traits define higher dimensional arrays.

```
Array1(I, E): trait
  introduces
     __[__]: Array[I, E], I    → E
     assign: Array[I, E], I, E → Array[I, E]
     const:  E                 → Array[I, E]
  asserts with a, a1, a2: Array[I, E], i, j: I, e: E
    assign(a, i, e)[j] = (if i = j then e else a[j]);
    const(e)[i] = e;
    ∀ i (a1[i] = a2[i]) ⇒ a1 = a2

Array2(I, J, E): trait
  introduces
     __[__, __]: Array[I, J, E], I, J    → E
     assign:     Array[I, J, E], I, J, E → Array[I, J, E]
     const:      E                       → Array[I, J, E]
  asserts with a, a1, a2: Array[I, J, E], i, i1: I, j, j1: J, e: E
    assign(a, i, j, e)[i1, j1] = (if i = i1 ∧ j = j1 then e else a[i1, j1]);
    const(e)[i, j] = e;
    ∀ i ∀ j (a1[i, j] = a2[i, j]) ⇒ a1 = a2
```

A.9 Finite mappings

The following trait defines one- and two-dimensional mappings. Similar traits define higher dimensional mappings. Mappings differ from arrays in that they may not be defined for all values of their indices (the **generated by** axiom ensures that mappings are defined for only finitely many values of their indices).

```
Mapping1(D, R): trait
  introduces
    empty:                      → Map[D, R]
    update:   Map[D, R], D, R → Map[D, R]
    __[__]:   Map[D, R], D    → R
    defined:  Map[D, R], D    → Bool
  asserts with m: Map[D, R], d, d1, d2: D, r: R
    sort Map[D, R] generated by empty, update;
    sort Map[D, R] partitioned by defined, __[__];
    update(m, d2, r)[d1] = (if d1 = d2 then r else m[d1]);
    ¬defined(empty, d);
    defined(update(m, d2, r), d1) ⇔ d1 = d2 ∨ defined(m, d1)
  implies with d: D
    converts __[__], defined exempting empty[d]

Mapping2(D1, D2, R): trait
  introduces
    empty:                                  → Map[D1, D2, R]
    update:      Map[D1, D2, R], D1, D2, R → Map[D1, D2, R]
    __[__, __]:  Map[D1, D2, R], D1, D2    → R
    defined:     Map[D1, D2, R], D1, D2    → Bool
  asserts with m: Map[D1, D2, R], d1, d1′: D1, d2, d2′: D2, r: R
    sort Map[D1, D2, R] generated by empty, update;
    sort Map[D1, D2, R] partitioned by defined, __[__,__];
    update(m, d1, d2, r)[d1′, d2′] =
      (if d1 = d1′ ∧ d2 = d2′  then r else m[d1′, d2′]);
    ¬defined(empty, d1, d2);
    defined(update(m, d1, d2, r), d1′, d2′) ⇔
      (d1 = d1′ ∧ d2 = d2′) ∨ defined(m, d1′, d2′)
  implies with d1: D1, d2: D2
    converts __[__,__], defined exempting empty[d1, d2]
```

A.10 Finite multisets

```
Multiset(E): trait
  includes Integer
  introduces
    {}: → Mset[E]
    {__}: E → Mset[E]
    insert, delete: E, Mset[E] → Mset[E]
    __∈ __: E, Mset[E] → Bool
    __∪ __, __∩ __, __-__: Mset[E], Mset[E] → Mset[E]
    __⊂ __, __⊃ __, __⊆ __, __⊇ __: Mset[E], Mset[E] → Bool
    size: Mset[E] → Int
    count: E, Mset[E] → Int
  asserts with s, s1, s2: Mset[E], e, e1, e2: E
   % Basic axioms: induction axiom, extensionality
     sort Mset[E] generated by {}, insert;
     ∀ e (count(e, s1) = count(e, s2)) ⇔ s1 = s2;
   % Recursive definitions of size, count
     size({}) = 0;
     size(insert(e, s)) = size(s) + (if e ∈ s then 0 else 1);
     count(e, {}) = 0;
     count(e, insert(e1, s)) = count(e, s) + (if e = e1 then 1 else 0);
   % Explicit definitions
     e ∈ s ⇔ count(e, s) > 0;
     {e} = insert(e, {});
     s1 ⊆ s2 ⇔ ∀ e (count(e, s1) ≤ count(e, s2));
     s1 ⊂ s2 ⇔ s1 ⊂ s2 ∧ s1 ≠ s2;
     s1 ⊇ s2 ⇔ s2 ⊆ s1;
     s1 ⊃ s2 ⇔ s2 ⊂ s1;
   % Extensional definitions
     count(e, delete(e1, s)) = count(e, s) - (if e = e1 then 1 else 0);
     count(e, s1 ∪ s2) = count(e, s1) + count(e, s2);
     count(e, s1 ∩ s2) = min(count(e, s1), count(e, s2));
     count(e, s1 - s2) = count(e, s1) - count(e, s2);
  implies with e: E, s: Mset[E]
    s = {} ⇔ ∀ e ¬(e ∈ s);
    e ∈ insert(e, s);
    ¬(e ∈ delete(e, s));
    s ∪ {} = s;
    s ∩ {} = {};
    delete (e, s) ⊆ s;
    s ⊆ insert(e, s);
    size(s) = size(delete(e, s)) + (if e ∈ s then 1 else 0)
```

A.11 Null elements

```
Null(T): trait
  introduces
    nil:              → Null[T]
    embed:  T         → Null[T]
    __.val: Null[T]   → T
  asserts with t: T
    sort Null[T] generated freely by nil, embed;
    embed(t).val = t
  implies with t, t1, t2: T
    embed(t1) = embed(t2) ⇔ t1 = t2;
    embed(t) ≠ nil
```

A.12 Finite sequences

```
Sequence(E): trait
  includes Integer
  introduces
    {}:                              → Seq[E]
    __⊢__:        Seq[E], E          → Seq[E]
    __⊣__:        E, Seq[E]          → Seq[E]
    __‖__:        Seq[E], Seq[E]     → Seq[E]
    __∈ __:       E, Seq[E]          → Bool
    head, last:   Seq[E]             → E
    tail, init:   Seq[E]             → Seq[E]
    len:          Seq[E]             → Int
    __[__]:       Seq[E], Int        → E
  asserts with e, e1, e2: E, s, s1, s2: Seq[E], n: Int
      sort Seq[E] generated by {}, ⊢;
      e ⊣ {} = {} ⊢ e;
      (e1 ⊣ s) ⊢ e2 = e1 ⊣ (s ⊢ e2);
      s ‖ {} = s;
      s1 ‖ (s2 ⊢ e) = (s1 ‖ s2) ⊢ e;
      ¬(e ∈ {});
      e ∈ (s ⊢ e1) ⇔ (e ∈ s) ∨ e = e1;
      head(e ⊣ s) = e;
      tail(e ⊣ s) = s;
      init(s ⊢ e) = s;
      last(s ⊢ e) = e;
      len({}) = 0;
      len(s ⊢ e) = succ(len(s));
      s[0] = head(s);
      n ≥ 0 ⇒ s[n+1] = tail(s)[n]
   implies with e, e1, e2: E, s, s1, s2, s3: Seq[E]
       sort Seq[E] generated by {}, ⊣:E,Seq[E]→Seq[E];
       e ⊣ s ≠ {};
       s ⊢ e ≠ {};
       s ≠ {} ⇒ s = head(s) ⊣ tail(s);
       s ≠ {} ⇒ s = init(s) ⊢ last(s);
       e ∈ (e ⊣ s);
       s ≠ {} ⇒ head(s) ∈ s ∧ last(s) ∈ s
```

A.13 Finite sets

```
Set(E): trait
  includes Integer
  introduces
    {}: → Set[E]
    {__}: E → Set[E]
    insert, delete: E, Set[E] → Set[E]
    __∈ __: E, Set[E] → Bool
    __∪ __, __∩ __, __-__: Set[E], Set[E] → Set[E]
    __⊂ __, __⊃ __, __⊆ __, __⊇ __: Set[E], Set[E] → Bool
    size: Set[E] → Int
  asserts with s, s1, s2: Set[E], e, e1, e2: E
   % Basic axioms: induction axiom, extensionality
     sort Set[E] generated by {}, insert;
     ∀ e (e ∈ s1 ⇔ e ∈ s2) ⇔ s1 = s2;
   % Recursive definitions of ∈, size
     ¬(e ∈ {});
     e1 ∈ insert(e2, s) ⇔ e1 = e2 ∨ e1 ∈ s;
     size({}) = 0;
     size(insert(e, s)) = size(s) + (if e ∈ s then 0 else 1);
   % Explicit definitions
     {e} = insert(e, {});
     s1 ⊆ s2 ⇔ ∀ e (e ∈ s1 ⇒ e ∈ s2);
     s1 ⊂ s2 ⇔ s1 ⊂ s2 ∧ s1 ≠ s2;
     s1 ⊇ s2 ⇔ s2 ⊆ s1;
     s1 ⊃ s2 ⇔ s2 ⊂ s1;
   % Extensional definitions
     e ∈ delete(e1, s) ⇔ e ≠ e1 ∧ e ∈ s;
     e ∈ (s1 ∪ s2) ⇔ e ∈ s1 ∨ e ∈ s2;
     e ∈ (s1 ∩ s2) ⇔ e ∈ s1 ∧ e ∈ s2;
     e ∈ (s1 - s2) ⇔ e ∈ s1 ∧ ¬(e ∈ s2);
  implies with e: E, s: Set[E]
    s = {} ⇔ ∀ e ¬(e ∈ s);
    e ∈ insert(e, s);
    ¬(e ∈ delete(e, s));
    s ∪ {} = s;
    s ∩ {} = {};
    delete (e, s) ⊆ s;
    s ⊆ insert(e, s);
    size(s) = size(delete(e, s)) + (if e ∈ s then 1 else 0)
```

B Software tools for IOA

A variety of software tools support the description and analysis of distributed systems using IOA. The following tools are available for download from `http://theory.lcs.mit.edu/tds/ioa`.

- `ioaCheck`: a syntax and static semantic checker, which can be used to check the well-formedness of descriptions for I/O automata or as a prettyprinter to tidy up descriptions of I/O automata. (See Appendix B.1.)
- `ioaSim`: a simulator, for testing the behavior of I/O automata. (See [9].)
- An LSL data type library, which supplies specifications of the data types built into IOA (see Appendix A, as well as of other common abstract data types for use in describing I/O automata. The LSL Handbook [8] forms the basis for this library. Users can extend the library by writing additional LSL specifications (see Section 9 and Part IV).
- `ioa2lsl`: a translator from IOA into LSL, which produces formal proof obligations for verifying invariants and simulation relations that are asserted in IOA specifications. `ioa2lsl` is intended for use with the Larch Prover [5]. (See Appendix B.3.)
- `ioa2isabelle`: a similar interface between IOA and Isabelle [12], which also produces formal proof obligations for verifying invariants and simulation relations.

B.1 `ioaCheck`: a static semantic checker for IOA and LSL

`ioaCheck` checks the syntax and static semantics of IOA specifications and LSL traits. It is invoked by a command line of the form

```
ioaCheck [option ...] source-file ...
```

The names of the source files should end with either `.ioa` or `.lsl`. If no options are provided on the command line, `ioaCheck` simply reports any errors that it finds in the source files. If options are provided, they produce the following results.

- `-p` causes `ioaCheck` to prettyprint (on the standard output) the IOA source files named on the command line. `ioaCheck` will prettyprint an LSL source file only if it is the first source file named on the command line. Prettyprinting indents IOA and LSL specifications to reveal their structure, and it breaks lines that exceeds the margin.
- `-path` *directoryList* instructs `ioaCheck` to search the directories in the colon-separated *directoryList* for additional source files. The default value of *directoryList* is `'.'`, which instructs `ioaCheck` to search only the current working directory. Whenever `ioaCheck` needs the specification of an LSL trait `T`, it searches for it in a file named `T.lsl`. Whenver it needs the specification of an I/O automaton named `A`, but cannot find that specification in one of the source files listed on the command line, it searches for it in a file named `A.ioa`.
- `-il` causes `ioaCheck` to translate the IOA and LSL source files named on the command line into an intermediate form (written to the standard output) suitable for use with other IOA tools such as the simulator. See Appendix B.2 for a description of this intermediate form.

B.2 Intermediate language

We describe syntax of the intermediate language for IOA and LSL in the same fashion as we described the syntax of IOA and LSL (see Section 12). The lexical grammar of the intermediate language uses the following symbols:

- Punctuation marks: `(` `)`
- Keywords: **actions**, **actuals**, **all**, **apply**, **asserts**, **assign**, **assumes**, **automaton**, **axioms**, **backward**, **case**, **choose**, **compose**, **const**, **converts**, **det**, **eff**, **ensuring**, **enum**, **exempting**, **exists**, **fire**, **for**, **formals**, **forward**, **generated**, **generatedFreely**, **hidden**, **id**, **if**, **ignore**, **implies**, **infix**, **initially**, **input**, **internal**, **invariant**, **ioa**, **lit**, **locals**, **mixfix**, **ops**, **output**, **partitioned**, **postfix**, **pre**, **prefix**, **proof**, **ref**, **rename**, **schedule**, **scope**, **select**, **sim**, **sim_entry**, **sim_fire**, **sorts**, **states**, **subscope**, **task**, **tasks**, **trait**, **traitref**, **traits**, **transitions**, **tuple**, **union**, **using**, **vars**, **where**, **while**, **with**, **yield**
- `IDENTIFIER`s and `OPERATOR`s, as for IOA and LSL.
- `NUMBER`s, which consist of sequences of digits. `SORTID`s for sorts, which consist of the letter `s` followed by a `NUMBER`
- `OPID`s for operators, which consist of the letters `op` followed by a `NUMBER`.
- `VARID`s for variables, which consist of the letter `v` followed by a `NUMBER`.
- `ACTIONID`s for actions, which consist of the letter `a` followed by a `NUMBER`.
- `TRANSITIONID`s for transition definitions, which consist of the letter `t` followed by a `NUMBER`.

Syntax of intermediate language specifications

```
specification    ::= '(' 'ioa' decls specUnit* ')'
specUnit         ::= automaton | assertion | trait | ioaAxioms | shorthand
ioaAxioms        ::= '(' 'axioms' traitRef ')'
```

Syntax of sort, operator, and variable declarations in the intermediate language

```
decls            ::= '(' sorts ops vars ')'
sorts            ::= '(' 'sorts' sort* ')'
sort             ::= '(' SORTID IDENTIFIER '(' sortId* ')' 'lit'? ')'
ops              ::= '(' 'ops' op* ')'
op               ::= '(' OPID operator signature ')'
operator         ::= constant | numeral | plainOp | ifOp | infixOp
                       | mixfixOp | prefixOp | postfixOp | selectOp
constant         ::= '(' 'id' IDENTIFIER ')'
numeral          ::= '(' 'const' NUMBER ')'
plainOp          ::= '(' 'id' IDENTIFIER ')'
ifOp             ::= '(' 'if' ')'
infixOp          ::= '(' 'infix' OPERATOR ')'
mixfixOp         ::= '(' 'mixfix' NUMBER trueFalse trueFalse
                         openSym closeSym ')'
```

```
postfixOp       ::= '(' 'postfix' OPERATOR ')'
prefixOp        ::= '(' 'prefix' OPERATOR ')'
selectOp        ::= '(' 'select' IDENTIFIER ')'
trueFalse       ::= 'true' | 'false'
signature       ::= '(' SORTID* ')' SORTID
vars            ::= '(' 'vars' var* ')'
var             ::= '(' VARID IDENTIFIER SORTID ')'
```

All sorts, operators, and variables in an intermediate language specification are enumerated in its `decls` section, which associates an internal `SORTID`, `OPID`, or `VARID` with each external `IDENTIFIER` or `OPERATOR`. The keyword `lit` indicates a sort with contants that are decimal literals. See Section 21 for definitions of `openSym` and `closeSym`.

Syntax of automaton definitions in the intermediate language

```
automaton       ::= '(' 'automaton' automatonName formals? automatonDef ')'
automatonName   ::= IDENTIFIER
formals         ::= '(' 'formals' formal+ ')'
formal          ::= term
automatonDef    ::= primitive | composition
```

Syntax of primitive automaton definitions in the intermediate language

```
primitive       ::= '(' actions states transitions? tasks? schedule?  ')'
actions         ::=  '(' 'actions' action+ ')'
action          ::= '(' ACTIONID actionType IDENTIFIER (formals where?)?
                        extension ')'
actionType      ::= 'input' | 'output' | 'internal'
where           ::= '(' 'where' term ')'
```

Syntax of state definitions in the intermediate language

```
states          ::= '(' 'states' state* initially? ')'
state           ::= '(' VARID value? ')'
value           ::= term | choice
initially       ::= '(' 'initially' term ')'
```

Syntax of transition definitions in the intermediate language

```
transitions     ::= '(' 'transitions' transition+ ')'
transition      ::= '(' TRANSITIONID caseName ACTIONID actionActuals? locals?
                        where? precondition? effect? ')'
caseName        ::= '(' 'case' IDENTIFIER? ')'
actionActuals   ::= '(' 'actuals' term+ ')'
locals          ::= '(' 'locals' VARID+ ')'
precondition    ::= '(' 'pre' predicate ')'
predicate       ::= term
effect          ::= program ensuring?
program         ::= '(' statement* ')'
```

```
statement       ::=  assignment | conditional | loop | simStatement
assignment      ::= '(' 'assign' lvalue value ')'
lvalue          ::= VARID | '(' 'apply' OPID lvalue term+ ')'
choice          ::= '(' 'choose' {'(' VARID term? ')' yieldprogram?}? ')'
conditional     ::= '(' 'if' '(' thenClause+ ')' program ')'
thenClause      ::= '(' predicate program ')'
loop            ::= '(' 'for' VARID term program ')'
ensuring        ::= '(' 'ensuring' predicate ')'
```

Syntax of task definitions in the intermediate language

```
tasks           ::= '(' 'tasks' task+ ')'
task            ::= '(' 'task' '(' actionSet+ ')' forClause? ')'
actionSet       ::= IDENTIFIER actionActuals where?
forClause       ::= '(' 'for' VARID+ ')'
```

Syntax of composite automaton definitions in the intermediate language

```
composition     ::= '(' components hidden? ')' primitive?
components      ::= '(' 'compose' component+ ')'
component       ::= '(' IDENTIFIER automatonActuals? term ')'
automatonActuals ::= '(' 'actuals' term+ ')'
hidden          ::= '(' 'hidden' actionSet+ ')'
```

Syntax of invariants and simulation relations in the intermediate language

```
assertion       ::= invariant | simulation
invariant       ::= '(' 'invariant' invariantName automatonName predicate ')'
invariantName   ::= IDENTIFIER
simulation      ::= '(' 'sim' direction automatonName automatonName
                       predicate simDetails? ')'
direction       ::= 'forward' | 'backward'
simDetails      ::= '(' 'proof' simStates '(' stateMap* ')'
                        '(' stepMap* ')' ')'
simStates       ::= '(' 'states' state* ')'
stateMap        ::= '(' term value ')'
stepMap         ::= '(' 'sim_entry' TRANSITIONID stepFormals program ')'
stepFormals     ::= '(' 'formals' formal* ')'
```

Syntax of simulator commands in the intermediate language

```
simStatement    ::= ndrfire | ndrwhile | simfire
ndrfire         ::= '(' 'fire' ( TRANSITIONID actionActuals? )? ')'
ndrwhile        ::= '(' 'while' predicate program ')'
simfire         ::= '(' 'sim_fire' TRANSITIONID actionActuals?
                        ( 'using' '(' variable term ')'+ )? ')'
schedule        ::= '(' 'schedule' simStates ndrprogram ')'
ndrprogram      ::= '(' ndrStatement* ')'
ndrstatement    ::= assignment | conditional | ndrfire | ndrwhile
```

```
yieldprogram      ::= '(' 'det' yieldstatement* ')'
yieldstatement    ::= conditional | ndrwhile | ndryield
ndryield          ::= '(' 'yield' term ')'
```

Syntax of trait definitions in the intermediate language

```
trait             ::= '(' 'trait' traitName traitFormals?
                          ( shorthand | traitRef )* opDcls?
                          traitAxioms? consequences? ')'
traitName         ::= IDENTIFIER
traitFormals      ::= '(' ( SORTID | OPID )+ ')'
```

Syntax of axioms in the intermediate language

```
traitAxioms       ::= '(' 'asserts' varDcls? prop+ extension ')'
varDcls           ::=
prop              ::= term | genPartBy | traitRef | converts
term              ::= applicationTerm | quantifiedTerm
                        | referenceTerm | literal
applicationTerm   ::= '(' 'apply' OPID term+ ')'
quantifiedTerm    ::= '(' quantifier term ')'
quantifier        ::= ( 'all' | 'exists' ) VARID
referenceTerm     ::=   VARID | OPID | SORTID
literal           ::= '(' 'lit' SORTID NUMBER ')'
genPartBy         ::= '(' genPartKind SORTID OPID+ ')'
genPartKind       ::= 'generated' | 'generatedFreely' | 'partitioned'
```

Syntax of shorthands in the intermediate language

```
shorthand         ::= enumeration | tuple | union
enumeration       ::= '(' 'enum' SORTID OPID enumItem+ ')'
enumItem          ::= '(' '"' IDENTIFIER '"' OPID ')'
tuple             ::= '(' 'tuple' SORTID OPID field+ ')'
field             ::= '(' '"' IDENTIFIER '"' SORTID OPID OPID ')'
union             ::= '(' 'union' SORTID SORTID OPID field+ ')'
```

The `OPID` in an `enumeration` is the successor operator for the elements of the enumerated sort identified by the `SORTID`, which has elements named by the `IDENTIFIER`s and identified by the `OPID`s in the `enumItem`s.

The `OPID` in a `tuple` names the operator that constructs elements of the tuple sort identified by the `SORTID`. Each `field` in the tuple is named by the given `IDENTIFIER`, has the sort identified by `SORTID`, and has *get* and *set* operators identified by the two `OPID`s.

The two `SORTID`s in a `union` name the union sort and its associated tag sort. The `OPID` names the tag operator. Each `field` in the union is named by the given `IDENTIFIER`, has the sort identified by `SORTID`, and has *to* and *from* operators identified by the two `OPID`s.

Syntax of trait references in the intermediate language

```
traitRef          ::= '(' 'traitref' '"' traitName '"' '(' traitActual* ')'
```

```
                          '(' constructorDef? ')' ')'
traitActual      ::= SORTID | OPID
constructorDef   ::= IDENTIFIER NUMBER '.' NUMBER
```

A `constructorDef` can appear in a `traitRef` only when that `traitRef` is part of an `ioaAxioms`, in which case each `traitActual` is a `SORTID`. The `IDENTIFIER` in the `constructorDef` is associated with a n-ary sort constructor defined by the given `traitName`, where n is the number of `traitActuals`, which must equal the value of the first `NUMBER` in the `constructorDef`.[15]

Syntax of consequences in the intermediate language

```
consequences     ::= '(' 'implies' varDcls? prop+ extension ')'
```

Syntax of conversions in the intermediate language

```
converts         ::= '(' 'converts' '(' OPID+ ')' exemption? ')'
exemption        ::= '(' 'exempting' term+ ')'
')'
```

Editorial note: Extra syntax to insert above.

```
renaming         ::= '(' 'rename' replace+ ')'
renamingMap      ::= replace*
replace          ::= '(' traitParam traitParam ')'
operator         ::= literal | OPID
sortConstructor  ::= SORTID
```

B.3 `ioa2lsl` and `ioa2isabelle`: interfaces to theorem provers

[15] Editorial note: The definition of the intermediate language may be changed by replacing the `IDENTIFIER` in a `constructorDef` with a `SORTID` and by eliminating the dot and the second §NUMBER, which currently is always 0.

C Bibliography

References

[1] Michel Bidoit. *Pluss, un langage pour le développement de spécifications algébriques modulaires.* Thése d'Etat, Université Paris-Sud, Orsay, May 1989.

[2] K. Mani Chandy and Jayadev Misra. *Parallel Program Design.* Addison Wesley, 1988.

[3] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics.* EATCS Monographs on Theoretical Computer Science, Vol. 6. Springer-Verlag, 1985.

[4] Herbert B. Enderton, *A Mathematical Introduction to Logic, Second Edition*, Academic Press, 2000.

[5] Stephen J. Garland and John V. Guttag. "A guide to LP, the Larch Prover." TR-82, DEC Systems Research Center, 1991. Updated version available as `http://www.sds.lcs.mit.edu/Larch/LP/overview.html`.

[6] Stephen J. Garland and Nancy A. Lynch, "Using I/O automata for developing distributed systems," *Foundations of Component-Based Systems*, Gary T. Leavens and Murali Sitaraman (ed.), Cambridge University Press, 2000.

[7] J. A. Goguen, J. W. Thatcher and E. G. Wagner. "An initial algebra approach to the specification, correctness, and implementation of abstract data types." *Current Trends in Programming Methodology IV: Data Structuring*, Raymond T. Yeh (ed.). Prentice-Hall, 1978.

[8] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[9] Dilsun Kırlı Kaynar, Anna Chefter, Laura Dean, Stephen Garland, Nancy Lynch, Toh Ne Win, and Antonio Ramírez-Robredo, "The IOA simulator," Technical Report MIT/LCS/TR-843, MIT Laboratory for Computer Science, 2002.

[10] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[11] Nancy A. Lynch and Mark Tuttle. "Hierarchical correctness proofs for distributed algorithms." Technical Report MIT/LCS/TR-387, MIT Laboratory for Computer Science, 1987.

[12] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, Lecture Notes in Computer Science, Vol. 2283, Springer-Verlag, 2002.

[13] Jørgen A. Sogaard-Andersen, Stephen J. Garland, John V. Guttag, Nancy A. Lynch, and Anna Pogosyants. "Computer-assisted simulation proofs." *4th Conference on Computer Aided Verification*, 1993.

[14] Joshua A. Tauber and Stephen J. Garland, "Definition and expansion of composite automata in IOA," MIT Computer Science and Artificial Intelligence Laboratory, 2003, in progress.

[15] M. Wand. "Final algebra semantics and data type extensions." *Journal of Computer and System Sciences*, August 1979.