IOA: A Language for Specifying, Programming, and Validating Distributed Systems\textsuperscript{1}

Stephen J. Garland, Nancy A. Lynch, and Mandana Vaziri
MIT Laboratory for Computer Science\textsuperscript{2}

October 5, 2001

\textsuperscript{1}Editorial note: Notes in this document allude to potential changes in this document, as well as in the IOA language. Additional details concerning the formal semantics of IOA, plus references to papers about IOA, will be incorporated into this document.

\textsuperscript{2}Research supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-92-J-1795 and by Hanscom Air Force Base under contract F19628-95-C-0018, by the National Science Foundation under grants CCR-9504248 and CCR-9225124, and by the Air Force Office of Scientific Research and the Office of Naval Research under contract F49620-94-1-0199.
## Contents

I  IOA Tutorial  

1  Introduction  
   1.1  I/O automata  
   1.2  Executions and traces  
   1.3  Operations on automata  
   1.4  Properties of automata  

2  Using IOA to formalize descriptions of I/O automata  

3  Data types in IOA descriptions  

4  IOA descriptions for primitive automata  
   4.1  Automaton names and parameters  
   4.2  Action signatures  
   4.3  State variables  
   4.4  Transition relations  
      4.4.1  Transition parameters  
      4.4.2  Preconditions  
      4.4.3  Effects  
      4.4.4  Choose parameters  
   4.5  Tasks  

5  IOA notations for operations on automata  
   5.1  Composition  
   5.2  Specialization  
   5.3  Hiding output actions in a composition  

6  IOA descriptions of properties of automata  
   6.1  Invariants  
   6.2  Simulation relations  

II  IOA Data Types  

7  Built-in primitive types  
   7.1  Booleans  
   7.2  Integers  
   7.3  Natural numbers  
   7.4  Real numbers  
   7.5  Characters  
   7.6  Strings  


8    Built-in type constructors
     8.1 Arrays .............................................. 23
     8.2 Finite sets ........................................... 23
     8.3 Multisets ............................................. 24
     8.4 Sequences ............................................ 24
     8.5 Mappings ............................................. 24

9    Data type semantics
     9.1 Axiomatic specifications ................................ 25
     9.2 Axiom schemes ......................................... 27
     9.3 Combining LSL specifications ........................... 28
     9.4 Renaming sorts and operators in LSL specifications  ...... 28
     9.5 Stating intended consequences of LSL specifications ...... 29
     9.6 Recording assumptions in LSL specifications .......... 31
     9.7 Built-in operators and overloading ..................... 32
     9.8 Shorthands ............................................ 33

10   User-defined data types .............................. 34

III   IOA Reference Manual ................................ 37

11   Logical preliminaries ................................... 37
     11.1 Syntax ............................................... 37
     11.2 Semantics ............................................ 38
     11.3 Further terminology .................................. 39

12   Lexical syntax ........................................... 39

13   Automaton definitions .................................. 40

14   Type and type constructor definitions ............... 41

15   Primitive automata ...................................... 42
     15.1 Primitive automaton definitions .................... 42
     15.2 Automaton states .................................... 43
     15.3 Automaton transitions .............................. 44
     15.4 Automaton tasks .................................... 46

16   Operations on automata .................................. 47

17   Statements about automata ............................. 48

IV    LSL Reference Manual ................................ 50

18   Lexical syntax ........................................... 50

19   Traits ..................................................... 50
20 Sort and operator declarations 50
21 Axioms 52
22 Shorthands for sorts 53
23 Trait references 54
24 Consequences 55
25 Converts 56

V Appendices 57
A Axioms for built-in data types 57
B Software tools for IOA 58
C Bibliography 59
Part I

IOA Tutorial

The Input/Output (I/O) automaton model, developed by Nancy Lynch and Mark Tuttle [9], models components in asynchronous concurrent systems as labeled transition systems. Lynch’s book, *Distributed Algorithms* [8], 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 [7] 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 10 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: A process](image)

A typical example\(^1\) of an I/O automaton is a process in an asynchronous distributed system. Figure 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* [8], 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 \( \text{init}(v)_i \), each of which represents the receipt of an input value \( v \), and it can produce outputs of the form \( \text{decide}(v)_i \), 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 \( \text{send}(m)_{i,j} \), each of which represents sending a message \( m \) to some process named \( P_j \), and input actions of the form \( \text{receive}(m)_{i,j} \), 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 2: A channel](image)

Another example of an I/O automaton is a FIFO message channel. Figure 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 \( \text{send}(m)_{i,j} \), and its output actions have the form \( \text{receive}(m)_{i,j} \).

Process and channel automata can be composed as shown in Figure 3, by matching the output actions of one automaton with the input actions of another. Thus, each output action \( \text{send}(m)_{i,j} \) of a process automaton is matched and performed together with an input action \( \text{send}(m)_{i,j} \) of some channel automaton, and each input action \( \text{receive}(m)_{i,j} \) of a process automaton is matched and performed together with an output action \( \text{receive}(m)_{i,j} \) 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 \( \pi \) is said to be **enabled** in a state \( s \) if there is another state \( s' \) such that \((s, \pi, s')\) 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 3: Composing channel and process automata
1.2 Executions and traces

An execution fragment of an I/O automaton is either a finite sequence $s_0, \pi_1, s_1, \pi_2, \ldots, \pi_n, s_n$, or an infinite sequence $s_0, \pi_1, s_1, \pi_2, \ldots$, of alternating states $s_i$ and actions $\pi_i$ 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 $\pi$, so do all component automata that have $\pi$ 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, as follows.

For the purpose of the following definitions, 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 $(s, \pi, s')$ of $A$, there is an execution fragment $\alpha$ of $B$ starting with $u$, containing the same external actions as $\pi$, 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 $I_B$,

---

$^2$In some previous work such relations are called weak forward simulations.

$^3$In some previous work such relations are called weak backward simulations.
• 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 \( u, u' \) of \( B \) satisfying the invariants such that \( R(s, u) \), and for every step \( (s, \pi, s') \) of \( A \), there is an execution fragment \( \alpha \) of \( B \) ending with \( u' \), containing the same external actions as \( \pi \), 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 4 contains a simple IOA description for an automaton, \texttt{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 components. The signature consists of input actions \texttt{add(i, j)}, one for each pair of values of \( i \) and \( j \), and output actions \texttt{result(k)}, one for each value of \( k \). The type \texttt{Int}, used to represent integers, is a built-in type in IOA (see Section 7.2).

\begin{verbatim}
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 4: IOA description of an adder
\end{verbatim}

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

The transitions of the automaton \texttt{Adder} are given in precondition/effect style. The input action \texttt{add(i, j)} has no precondition, which is equivalent to its having \texttt{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 \texttt{add(i, j)} is to change \texttt{value} to the sum of \( i \) and \( j \) and to set \texttt{ready} to \texttt{true}. The output action \texttt{result(k)} can occur only when it is enabled, that is, only in states where its precondition \( k = \texttt{value} \& \texttt{ready} \) is true. Its effect is to set \texttt{ready} back to \texttt{false}. Traces of \texttt{Adder} are sequences of external actions such as
add(3, 2), result(5), add(1, 2), add(-1, 1), result(0), ...
that start with an add action, in which every result action returns the sum computed by the last
add action, and in which every pair of result actions must be separated by one or more add actions.

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

Figure 5: IOA description of a reliable communication channel

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

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

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

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

3 Data types in IOA descriptions

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

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

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

- Array[I, E] is an array of elements of type E indexed by elements of type I.

- Map[D, R] is a finite partial mapping of elements of a domain type D to elements of a range
type R. Mappings differ from arrays in that their domains are always finite, and in that they
may not be totally defined.
• Seq[E] is a finite sequence of elements of type E.

• Set[E] is a finite set of elements of type E.

• Mset[E] is a finite multiset of elements of type E.

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

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>Logical Operator</th>
<th>Datatype Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>Meaning</td>
</tr>
<tr>
<td>∀</td>
<td>For all</td>
</tr>
<tr>
<td>∃</td>
<td>There exists</td>
</tr>
<tr>
<td>¬</td>
<td>Not</td>
</tr>
<tr>
<td>≠</td>
<td>Not equals</td>
</tr>
<tr>
<td>∧</td>
<td>And</td>
</tr>
<tr>
<td>∨</td>
<td>Or</td>
</tr>
<tr>
<td>⇒</td>
<td>Implies</td>
</tr>
<tr>
<td>⇔</td>
<td>If and only if</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4 IOA descriptions for primitive automata

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

4.1 Automaton names and parameters

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

4.2 Action signatures

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

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

\[
\text{signature}
\]

\[
\text{input} \quad \text{add}(i, j: \text{Int}) \text{ where } i > 0 \land j > 0
\]

\[
\text{output} \quad \text{result}(k: \text{Int}) \text{ where } k > 1
\]

could have been used for the automaton \text{adder} to restrict the values of the input parameters to positive integers and the value of the output parameter to integers greater than 1.

4.3 State variables

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

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

\[
\text{automaton} \quad \text{Choice}
\]

\[
\text{signature}
\]

\[
\text{output} \quad \text{result}(i: \text{Int})
\]

\[
\text{states}
\]

\[
\text{num: Int := choose n where } 1 \leq n \land n \leq 3,
\]

\[
\text{done: Bool := false}
\]

\[
\text{transitions}
\]

\[
\text{output} \quad \text{result}(i)
\]

\[
\text{pre} \quad \neg \text{done} \land i = \text{num}
\]

\[
\text{eff} \quad \text{done := true}
\]

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

\footnote{Later versions of IOA may also allow us to parameterize automata by operations (e.g., ordering relations) on a data type.}
For example, in the automaton Choice (Figure 6), the state variable n is initialized nondeterministically to some value of the variable \( n \) that satisfies the predicate \( 1 \leq n \wedge n \leq 3 \), i.e., to one of the values 1, 2, or 3 (the value of \( n \) must be an integer because it is constrained to have the same type, Int, as the variable n to which it will be assigned). The automaton Choice can return the selected value at most once in an output action.

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

type cardIndex = enumeration of 1, 2, 3, \ldots, 52

automaton Shuffle

signature

internal swap(i, j: cardIndex)
output deal(a: Array[cardIndex, String])

states

dealt: Bool := false,
name: Array[cardIndex, String],
cut: cardIndex,
temp: String

so that \( \forall i: \text{cardIndex} \ (i \neq 52 \wedge i \neq \text{cut} \Rightarrow \text{name}[i] < \text{name}[\text{succ}(i)]) \)
\( \wedge \text{name}[52] < \text{name}[1] \)

transitions

internal swap(i, j)
pre \( \neg \text{dealt} \)

\( \text{eff temp} := \text{name}[i]; \)
\( \text{name}[i] := \text{name}[j]; \)
\( \text{name}[j] := \text{temp} \)

output deal(a)
pre \( \neg \text{dealt} \wedge a = \text{name} \)

\( \text{eff dealt := true} \)

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

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

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

\(^5\) At present, users must expand the \( \ldots \) in the definition of the type cardIndex by hand; IOA will eventually provide more convenient notations for integer subranges.
4.4 Transition relations

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

4.4.1 Transition parameters

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

\[ \text{input send}(m: M, \text{const } i, \text{const } j) \]

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

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

\[ \text{internal swap}(i, j) \text{ where } i \neq j \]

\[ \begin{align*}
\text{pre } & \neg \text{deal}t \\
\text{eff temp} & := \text{name}[i]; \\
& \text{name}[i] := \text{name}[j]; \\
& \text{name}[j] := \text{temp}
\end{align*} \]

\[ \text{internal swap}(i, i) \]

\[ \begin{align*}
\text{pre } & \neg \text{deal}t \\
\text{eff temp} & := \text{name}[i]; \\
& \text{name}[i] := \text{name}[i]; \\
& \text{name}[i] := \text{temp}
\end{align*} \]

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

4.4.2 Preconditions

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

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

4.4.3 Effects

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

If the effect is missing, then the transition has none; i.e., it leaves the state unchanged.
Using programs to specify effects  A program is a list of statements, separated by semicolons. Statements in a program are executed sequentially. There are three kinds of statements:

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

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

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

axioms Subsequence for Seq[___]

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

Figure 8: IOA description of a lossy communication channel

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

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

\texttt{Subsequence(E): trait}
\begin{verbatim}
  includes Sequence(E)
  introduces \( \preceq \): Seq[E], Seq[E] \rightarrow \text{Bool}
  asserts with e, e1, e2: E, s, s1, s2: Seq[E]
    \{\} \preceq s;
    \neg((s \triangleright e) \preceq \{\})
    (s1 \triangleright e1) \preceq (s2 \triangleright e2) \iff (s1 \triangleright e1) \preceq s2 \lor (s1 \preceq s2 \land e1 = e2)
\end{verbatim}

Figure 9: Auxiliary specification with recursive definition of subsequence operator

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

\textit{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 \texttt{if} followed by a predicate, the keyword \texttt{then}, and a program segment; it ends with the keyword \texttt{fi}. In between, there can be any number of \texttt{elseif} clauses (each of which contains a predicate, the keyword \texttt{then}, and a program segment), and there can be a final \texttt{else} clause (which also contains a program segment). Figure 10 illustrates the use of a conditional statement in defining an automaton that distributes input values into one of three sets. Section 8.2 describes the set data type and the operators \{\} and \texttt{insert}.

\textit{For statements} A \texttt{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 \texttt{for} followed by a variable, a clause describing a set of values for this variable, the keyword \texttt{do}, a program segment, and the keyword \texttt{od}.

Figure 11 illustrates the use of a \texttt{for} statement in a high-level description of a multicast algorithm. Its first line defines the \texttt{Packet} data type to consist of triples \{\texttt{contents}, \texttt{source}, \texttt{dest}\}, in which \texttt{contents} represents a message, \texttt{source} the \texttt{Node} from which the message originated, and \texttt{dest} the set of \texttt{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 \texttt{queue}, which represents, for each \texttt{Node}, the sequence of packets delivered to that \texttt{Node}, but not yet read by the \texttt{Node}.

The \texttt{mcast} action inserts a new packet in the network; the notation \{\texttt{m, i, I}\} is defined by the tuple data type (Section 9.8) and the \texttt{insert} operator by the multiset data type (Section 8.3). The \texttt{deliver} action, which is described using a \texttt{for} statement, distributes a packet to all nodes in its destination set (by appending the packet to the queue for each node in the destination set and
automaton Distribute
signature
  input get(i: Int)
states
  small: Set[Int] := {},
  medium: Set[Int] := {},
  large: Set[Int] := {},
  bound1: Int,
  bound2: Int
  so that bound1 < bound2
transitions
  input get(i)
    eff if i < bound1 then small := insert(i, small)
    elseif i < bound2 then medium := insert(i, medium)
    else large := insert(i, large)
    fi

Figure 10: Example of a conditional statement

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

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

Figure 11: Example showing use of a for statement
then deleting the packet from the network). The read action receives the contents of a packet at a particular node by removing that packet from the queue of delivered packets at that node.

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

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

```
input crash

  eff buffer := choose
  so that buffer' ⊆ buffer
```

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

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

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

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

4.4.4 Choose parameters

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

The automaton LossyBuffer represents a message channel that loses a message each time it transmits one. The state of the automaton consists of a multiset buff of messages of type m. The input action for the channel, get(m), simply adds the message m to buff. The output action, put(m), delivers m while dropping another message, given by the choose parameter n. The precondition
automaton LossyBuffer(M: type)
signature
input  get(m: M)
output put(m: M)
states
  buff: Mset[M] := {}
transitions
input  get(m)
  eff  buff := insert(m, buff)
output put(m)
  choose n: M
  pre  m \in buff \land n \in buff \land (m \neq n \lor count(n, buff) > 1)
  eff  buff := delete(m, delete(n, buff))

Figure 12: Example of the use of choose parameters

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

Choose parameters provide syntactic sugar for defining transitions. It is possible to define transitions without them by using explicit quantification. For example, the transition for the put action in Figure 12 can be rewritten as follows:
output put(m)
  pre \exists n: M (m \in buff \land n \in buff \land (m \neq n \lor count(m, buff) > 1))
  eff  buff := choose
      so that \exists n: M (m \in buff \land n \in buff \land (m \neq n \lor count(m, buff) > 1)
                        \land buff' = delete(m, delete(n, buff)))

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

4.5 Tasks

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

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

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

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

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

\texttt{tasks} \{\texttt{deliver}(p) \textbf{for} p: \texttt{Packet} \} \{\texttt{read}(m, j) \textbf{for} m: \texttt{Message} \} \textbf{for} j: \texttt{Node} for the Multicast automaton places the \texttt{read} actions for different nodes in different tasks, so that the execution of \texttt{read} actions for one node cannot starve execution of \texttt{receive} actions for another. The values of variables appearing in task definitions can be constrained further by \textbf{where} clauses following the \textbf{for} clauses.

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

5 IOA notations for operations on automata

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

5.1 Composition

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

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

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

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

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

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

\textsuperscript{6}Eventually IOA will also provide notations for renaming actions.
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)),
       leader(m: String, const i)
states
  pending: $set[String] := \{name(i)\},
  status: Status := waiting
transitions
  input receive(m, j, i)
    eff if m > name(i) then pending := insert(m, pending)
    elseif m = name(i) then status := elected
  fi
  output send(m, i, j)
  pre m ∈ pending
    eff pending := delete(m, pending)
  output leader(m, i)
  pre status = elected ∧ m = name(i)
    eff status := announced
tasks
  \{send(m, j, right(j)) for m: String, j: I\};
  \{leader(m, j) for m: String, j: I\}

Figure 13: IOA specification of election process

RingIndex(I, J): trait
introduces
  first: \rightarrow I
  left, right: I \rightarrow I
  name: I \rightarrow J
asserts with i, j: I
  sort I generated by first, right;
  \exists i \ (right(i) = first);
  right(i) = right(j) \iff i = j;
  left(right(i)) = i;
  name(i) = name(j) \iff i = j
implies with i: I
  right(left(i)) = i

Figure 14: Auxiliary specification for a finite ring of process identifiers

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

Figure 15: IOA specification of LCR algorithm
The full LCR leader election algorithm is described in Figure 15 as a composition of a set of process automata connected in a ring by reliable communication channels (Figures 2 and 5). The assumes statement on the first line repeats the assumption about the type \( i \) of process indices in Figure 13. The keyword components introduces a list of named components: one Process automaton, \( P[i] \), and one Channel automaton, \( C[i] \), for each element \( i \) of type \( I \). The component \( C[i] \) is obtained by instantiating the type parameters \( M \) and \( \text{Index} \) for the Channel automaton (Figure 5) 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 \( \text{send}(m, i, \text{right}(i)) \) of \( P[i] \) are identified with the input actions \( \text{send}(m, i, \text{right}(i)) \) of \( C[i] \), and the input actions \( \text{receive}(m, \text{left}(i), i) \) of \( P[i] \) are identified with the output actions \( \text{receive}(m, \text{left}(i), i) \) of \( C[\text{left}(i)] \), because \( \text{RingIndex} \) implies that right(\( \text{left}(i) \)) = \( i \). 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 16, 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.

\[
type abcd = \text{enumeration of } a, b, c, d
\]

axioms \( \text{RingIndex}(abcd, \text{String}) \)

automaton \( \text{LCR4} \)
  \( \text{components theOnly: LCR}(abcd) \)

Figure 16: IOA specification of four-process LCR algorithm

Even though the description of \( \text{LCR4} \) is not parameterized, it still defines a set of automata rather than single automaton: Figure 16 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 \( \text{name}(a) \), \( \text{name}(b) \), \( \text{name}(c) \), and \( \text{name}(d) \). 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 \( \text{LCR4} \), we can use a hidden statement, as in Figure 17.

automaton \( \text{LCR4a} \)
  \( \text{components theOnly: LCR4} \)
  hidden receive\((m, i, j)\), send\((m, i, j)\)

Figure 17: 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.

\[
\text{invariant of LCR: } P[i].\text{status} = \text{elected} \land P[j].\text{status} = \text{elected} \Rightarrow i = j
\]

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. When there is no ambiguity (i.e., when only one component has a state variable with a given name), the name of the automaton may be omitted.

6.2 Simulation relations

Simulation relations provide a convenient mechanism for showing that one automaton implements another, i.e., that every trace one is a trace of the other. In order to illustrate various simulation relations, we describe a modification, DelayedLossyChannel (Figure 18), of the LossyChannel (Figure 8) 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 \uplus [m, false]
        output remove(m)
            pre buffer \neq {} \land head(buffer).msg = m
            eff buffer := tail(buffer)
        input crash
            eff buffer := mark(buffer)
        internal lose
            eff buffer := choose
                so that subseqMarked(buffer', buffer)

Figure 18: Specification of an implementation of a lossy channel

The axioms statement in Figure 18 identifies a user-written specification (Figure 27) 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.

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:**

\[
\text{messages}(\text{DelayedLossyChannel.buffer}) = \text{LossyChannel.buffer}
\]

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:**

\[
\exists s: \text{Seq}[\text{MM}] \ (\text{subseqMarked}(s, \text{DelayedLossyChannel.buffer})
\land \text{LossyChannel.buffer} = \text{messages}(s))
\]

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. We list here the operators available for the built-in types; Appendix A defines their properties 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 primitive data types Bool, Int, Nat, Real, and Char 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 5, which is parameterized by the types M and Index.

- Compound data types formed using the type constructors Array, Set, MSet, Seq, and Map 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 22 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.

  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 operators \(\land\) and \(\lor\) bind more tightly than \(\Rightarrow\), which binds more tightly than \(\Leftrightarrow\).
### Operators for bool

<table>
<thead>
<tr>
<th>true, false</th>
<th>Sample input</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬</td>
<td>¬p</td>
<td>Negation (not)</td>
</tr>
<tr>
<td>∧, ∨</td>
<td>p ∧ q, p ∨ q</td>
<td>Conjunction (and), disjunction (or)</td>
</tr>
<tr>
<td>⇒</td>
<td>p ⇒ q</td>
<td>Implication (implies)</td>
</tr>
<tr>
<td>⇔</td>
<td>p ↔ q</td>
<td>Logical equivalence (if and only if)</td>
</tr>
</tbody>
</table>

#### 7.2 Integers

The integer data type, Int, provides constants and operators for the set of (positive and negative) integers.

<table>
<thead>
<tr>
<th>Operators for Int</th>
<th>Sample input</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 1, ...</td>
<td>123</td>
<td>Non-negative integers</td>
</tr>
<tr>
<td>-</td>
<td>-x</td>
<td>Additive inverse (unary minus)</td>
</tr>
<tr>
<td>abs</td>
<td>abs(x)</td>
<td>Absolute value</td>
</tr>
<tr>
<td>pred, succ</td>
<td>succ(x)</td>
<td>Predecessor, successor</td>
</tr>
<tr>
<td>+, -, *,</td>
<td>x + (y*z)</td>
<td>Addition, subtraction, multiplication</td>
</tr>
<tr>
<td>min, max</td>
<td>min(x, y)</td>
<td>Minimum, maximum</td>
</tr>
<tr>
<td>div, mod</td>
<td>mod(x, y)</td>
<td>Integer quotient, modulus</td>
</tr>
<tr>
<td>&lt;, ≤, &gt;, ≥</td>
<td>x ≤ y</td>
<td>Less (greater) than (or equal to)</td>
</tr>
</tbody>
</table>

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.

#### 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. 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 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.

#### 7.5 Characters

The character data type, Char, provides constants and operators for letters and digits.\(^7\)

\(^7\) Additional character constants will be provided in a future version of IOA.
### Operators for Char

<table>
<thead>
<tr>
<th>Operators for Char</th>
<th>Sample input</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A', ..., Z', a', ..., z', 0', ..., 9'</td>
<td>'J'</td>
<td>Letters and digits</td>
</tr>
<tr>
<td>≤, ≥</td>
<td>A' &lt;= Z'</td>
<td>ASCII ordering</td>
</tr>
</tbody>
</table>

#### 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 ≤, ≥, and ≥.

### 8 Built-in type constructors

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

#### 8.1 Arrays

The array data types, Array[I, E] and Array[I, I, E], provide constants and operators for one- and two-dimensional arrays of elements of some type E indexed by elements of some type I.

<table>
<thead>
<tr>
<th>Operators for Array[I, E]</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant(e)</td>
<td>Array with all elements equal to e</td>
</tr>
<tr>
<td>a[i]</td>
<td>Element indexed by i in array a</td>
</tr>
<tr>
<td>assign(a, i, e)</td>
<td>Array a' equal to a except that a'[i] = e</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operators for Array[I, I, E]</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant(e)</td>
<td>Array with all elements equal to e</td>
</tr>
<tr>
<td>a[i, j]</td>
<td>Element indexed by i, j in array a</td>
</tr>
<tr>
<td>assign(a, i, j, e)</td>
<td>Array a' equal to a except that a'[i, j] = e</td>
</tr>
</tbody>
</table>

The array (one- or two-dimensional) denoted by constant(e) is determined by context, as in constant(e)[i], or by an explicit qualification, as in constant(e):Array[I, I, E].

#### 8.2 Finite sets

The set data type, Set[E], provides constants and operators for finite sets of elements of some type E.

<table>
<thead>
<tr>
<th>Operators for Set[E]</th>
<th>Sample input</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>{}</td>
<td>Empty set</td>
</tr>
<tr>
<td>(...)</td>
<td>{e}</td>
<td>Set containing e alone</td>
</tr>
<tr>
<td>insert</td>
<td>insert(e, s)</td>
<td>Set containing e and all elements of s</td>
</tr>
<tr>
<td>delete</td>
<td>delete(e, s)</td>
<td>Set containing all elements of s, but not e</td>
</tr>
<tr>
<td>∈</td>
<td>e \in s</td>
<td>True iff e is in s</td>
</tr>
<tr>
<td>∪, ∩, -</td>
<td>(s \cup s') - (s \cap s')</td>
<td>Union, intersection, difference</td>
</tr>
<tr>
<td>⊂, ⊆, ⊇</td>
<td>s \subseteq s'</td>
<td>(Proper) subset (superset)</td>
</tr>
<tr>
<td>size</td>
<td>size(s)</td>
<td>Size (an Int) of s</td>
</tr>
</tbody>
</table>
8.3 Multisets

The multiset data type, \$\text{\$set(E)}\$, provides constants and operators for finite multisets of elements of some type E. Its operators are those for \$\text{Set(E)}\$, except that there is an additional operator \$\text{\$count(e, s)}\$ such that \$\text{\$count(e, s)}\$ is the number (an Int) of times an element \$e\$ occurs in a multiset \$s\$.

8.4 Sequences

The sequence data type, \$\text{Seq[E]}\$, provides constants and operators for finite sequences of elements of some type E.

<table>
<thead>
<tr>
<th>Operators for $\text{Seq[E]}$</th>
<th>Sample input</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>{}</td>
<td>Empty sequence</td>
</tr>
<tr>
<td>$\text{-}$</td>
<td>$s \text{-} e$</td>
<td>Sequence with $e$ appended to $s$</td>
</tr>
<tr>
<td>$\text{\textbackslash -}$</td>
<td>$e \text{-} s$</td>
<td>Sequence with $e$ prepended to $s$</td>
</tr>
<tr>
<td>$\text{</td>
<td></td>
<td>}$</td>
</tr>
<tr>
<td>$\text{&amp;}$</td>
<td>$e \text{&amp;} s$</td>
<td>True iff $e$ is in $s$</td>
</tr>
<tr>
<td>$\text{head, last}$</td>
<td>head(s)</td>
<td>First (last) element in sequence</td>
</tr>
<tr>
<td>$\text{init, tail}$</td>
<td>tail(s)</td>
<td>All but first (last) elements in sequence</td>
</tr>
<tr>
<td>$\text{len}$</td>
<td>len(s)</td>
<td>Length (an Int) of $s$</td>
</tr>
<tr>
<td>... [...]</td>
<td>$s[n]$</td>
<td>$n$th (an Int) element in $s$</td>
</tr>
</tbody>
</table>

8.5 Mappings

The mapping data type, \$\text{\$Map(D, R)}\$, provides constants and operators for finite partial mappings of elements of some domain type \$D\$ to elements of some range type \$R\$. Finite mappings differ from arrays in two ways: they may not be defined for all elements of \$D\$, and their domains are always finite.

<table>
<thead>
<tr>
<th>Operators for $\text{Map(D, R)}$</th>
<th>Sample input</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{empty}$</td>
<td>empty</td>
<td>Empty mapping</td>
</tr>
<tr>
<td>... [...]</td>
<td>empty</td>
<td>Image of $d$ under $m$</td>
</tr>
<tr>
<td>$\text{defined}$</td>
<td>defined(m, d)</td>
<td>True if $m[d]$ is defined</td>
</tr>
<tr>
<td>$\text{update}$</td>
<td>update(m, d, r)</td>
<td>Mapping $m'$ equal to $m$ except that $m'[d] = r$</td>
</tr>
</tbody>
</table>

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.\(^8\)

This section provides a tutorial introduction to LSL. It is taken from Chapter 4 of [7], 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 [7], 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 (such as C) or for a mathematical model of

\(^8\)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.
computation (such as 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.

Sequences(E): trait introduces

\{\}: \rightarrow \text{Seq}[E]  
\_\_ \_ \_ : E, \text{Seq}[E] \rightarrow \text{Seq}[E]  
last: \text{Seq}[E] \rightarrow E  
init: \text{Seq}[E] \rightarrow \text{Seq}[E]  

asserts with s: \text{Seq}[E], e: E

sort \text{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 \_\_\_\_\_1, s2: \text{Seq}[E], e1, e2: E

e_1 \_\_\_\_\_1 = e_2 \_\_\_\_\_2 \iff e_1 = e_2 \land s_1 = s_2;

e_1 \_\_\_\_\_1 \neq \{\}

Figure 19: Simplified LSL specification for sequences

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 19 describes some properties of finite sequences of elements of 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 property distinguishes sequences from sets, where it does not matter in which order elements are inserted.

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 a 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 20. 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., ∈).

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 20: A trait specifying some properties of sets

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. Every operator used in a trait must be declared; signatures are used to sort-check terms in 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 ≠ 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 \( \equiv (if \; and \; only \; if) \), which binds more tightly than
• the built-in conditional operator \( if \; -- \; then \; -- \; else \; -- \).

For example, the term \( p \equiv x \ast w \cdot a \cdot b = y \lor z \) can be written without parentheses and is equivalent to the fully parenthesized term \( p \equiv ((x + ((w \cdot a) \cdot b)) = y) \lor z \). 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 \( x \land y \land z \) is equivalent to \( (x \land y) \land z \), but \( x \land y \land z \) must be written as \( (x \lor y) \land z \) or as \( x \lor (y \land z) \), 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 \( \equiv \).

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 [6] or final algebras [14]. 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 \( \text{Nat} \) of natural numbers contains the integers 0, 1, 2, . . . and nothing else. A natural approach is to assert that the set \( \text{Nat} \) is the smallest set containing 0 and closed under the successor operation \( \text{succ} \) (defined by \( \text{succ}(n) = n+1 \)):

\[
\forall s: \text{Set}[\text{Nat}] \; (0 \in s \land \forall n: \text{Nat} \; (n \in s \Rightarrow \text{succ}(n) \in s) \Rightarrow \forall n: \text{Nat} \; (n \in s))
\]

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

There are several ways to remedy this problem. One is to posit some special, unaxiomatized relationship between the sort \( \text{Set}[E] \) and the sort \( E \) (i.e., that \( \text{Set}[E] \) 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 \( \text{Seq}[E] \) or \( \text{Map}[E,E] \) and the sort \( E \). Another approach, which avoids this problem, is to enlarge \( \text{Set0} \) with axioms like \( \exists s: \text{Set}[\text{Nat}] \land \forall n: \text{Nat} \; (n \in s) \) that force \( \text{Set}[E] \) 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. Similarly, 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.

A generated by statement justifies a induction schema for proving properties of a sort. For example, to prove \( \forall s : \text{Set}[E] \ (\text{size}(s) \geq 0) \) 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: \( \text{size}(\{\}) \geq 0 \)
- Induction step: \( \forall s : \text{Set}[E] \ \forall e : E \ (\text{size}(s) \geq 0 \Rightarrow \text{size}(\text{insert}(e, s)) \geq 0) \)

In general, a generated by statement is equivalent to an infinite set of formulas, one for each property (such as \( \text{size}(s) \geq 0 \)) that can be expressed in first-order logic.9

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 \( \text{size}(s) \geq 0 \). 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), 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, both the trait PartialOrder1 and the less structured trait PartialOrder2 in Figure 21 define a partial order to be an irreflexive, transitive order.

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

---

9. 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 \( \varepsilon \) asserts that terms indistinguishable by the observer \( \varepsilon \) 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 \_\_\_<\_\_: \text{T}, \text{T} \rightarrow \text{Bool}
  
  asserts with \text{x}: \text{T}
  
  \neg(x < x)

Transitive: trait
  
  introduces \_\_\_<\_\_: \text{T}, \text{T} \rightarrow \text{Bool}
  
  asserts with \text{x}, \text{y}, \text{z}: \text{T}
  
  \text{x} < \text{y} \land \text{y} < \text{z} \Rightarrow \text{x} < \text{z}

PartialOrder1: trait
  
  includes Irreflexive, Transitive

PartialOrder2: trait
  
  introduces \_\_\_<\_\_: \text{T}, \text{T} \rightarrow \text{Bool}
  
  asserts with \text{x}, \text{y}, \text{z}: \text{T}
  
  \neg(x < x);
  \text{x} < \text{y} \land \text{y} < \text{z} \Rightarrow \text{x} < \text{z}

Figure 21: Specifications of kinds of relations

can be used to make names coincide, to keep them from coinciding, or simply to replace them with more suitable names, as in

  includes Transitive(< for \text{\_\_\_<\_\_})

which we can use to assert that some operator other than < is transitive.

In general, a trait reference is a phrase \text{Tr(name1 for name2, \ldots)} that stands for the trait \text{Tr} with every occurrence of \text{name2} (which must be a sort, a sort constructor, or an operator) replaced by \text{name1}, etc. If \text{name2} is a sort or a sort constructor, this renaming changes the signatures of all operators in \text{Tr} in whose signatures \text{name2} appears. For example, the signature of the operator + changes to \text{Int,Seq[Int] \rightarrow Seq[Int]} in the trait reference includes Sequences(\text{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 19 makes the reference includes Sequences(\text{Int}) equivalent to includes Sequences(\text{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, specificers 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 \text{false} is illegal.

An implies clause makes claims about theory containment. Suppose we think that a consequence of the assertions of \text{Set0} is that the order in which elements are inserted in a set makes no difference. To formalize this claim, we could use the following clause to \text{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 they are part of. 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 ε
to Set₀ makes the claim that the trait’s axioms fully define the operator ε. This claim means that, if the interpretations of all the other operators are fixed, there is only one interpretation of ε 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 19, which define the meaning of last(s) and init(s) only when s ≠ {}. 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 the axioms. 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 not to be defined”). 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 specifications, preconditions for actions should ensure that their effects do not depend on the values of undefined terms. If an action has a nondeterministic effect, that effect should be specified using the choose operator or a so that clause. For example, the IOA specification

output pick1(x: Int, s: Set[Int])
  pre s ≠ {};
  eff x := choose e where e ∈ s
describes an action that is enabled for any pair (x, s) such that x ∈ s. Attempting to specify the action using an underspecified LSL operator will not produce the same result. For example, the IOA specification

output pick2(x: Int, s: Set[Int])
  pre s ≠ {};
  eff x := someElement(s)

describes an action that, for any nonempty set s, is enabled for exactly one pair (x, s), namely, (s, someElement(s)). A trait containing
asserts with s: Set[Int]
   s ≠ {} ⇒ someElement(s) ∈ s

does much more than constrain the value of someElement(s) to one appropriate for a choose operator: it constrains the value of someElement(s) to be the same each time that term is used in an IOA specification.

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:

```lsl
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.

```lsl
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 22: Preliminary specification of ordered sequences

Consider, however, specializing the Sequences trait to describe lexicographically ordered sequences, as in Figure 22. As written, OrderedSequences0 says nothing about whether the operator < defines an ordering over E; hence there is no reason to believe that the operator ≼ 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 23.

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 ≼ 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

```lsl
StringSequences1: trait
   includes String, OrderedSequences(String)
```
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 23: Specification of ordered sequences

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., is a subset of the theory associated with the trait String).

9.7 Built-in operators and overloading

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 overloading. 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 <), 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 <) explicitly, as in

\[(e1 \downarrow s1): Seq[E] < (e2 \downarrow s2): Seq[E] \iff \\
e1:E < e2:E \lor (e1:E = e2:E \land s1:Seq[E] < s2:Seq[E])
\]
\[t1:T < t2:T \lor (t1:T = t2:T \land s1:Seq[T] < s2:Seq[T])
\]

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

32
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 24.

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 24: 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 25. 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 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 25: 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 26. Each field name (e.g., $circ$) is incorporated in three distinct operators (e.g., $circ$→$\text{Figure\_tag}$, $circ$:$\text{Circle}$→$\text{Figure}$, and $\_\_\_circ$:$\text{Figure}$→$\text{Circle}$).

SampleUnion: trait

\begin{verbatim}
Figure\_tag
\end{verbatim}

enumeration of $sq$, $circ$

\begin{verbatim}
introduces
sq: Square \rightarrow \text{Figure}
circ: \text{Circle} \rightarrow \text{Figure}
\_\_\_sq: \text{Figure} \rightarrow \text{Square}
\_\_\_circ: \text{Figure} \rightarrow \text{Circle}
tag: \text{Figure} \rightarrow \text{Figure\_tag}
\end{verbatim}

\begin{verbatim}
asserts with s: \text{Square}, c: \text{Circle}
sort \text{Figure} generated by $sq$, $circ$;
sort \text{Figure partitioned} by $tag$, $\_\_\_sq$, $\_\_\_circ$;
tag($sq(s)$) = $sq$;
tag($circ(c)$) = $circ$;
$sq(s).sq = s$;
circ(c).circ = c
\end{verbatim}

Figure 26: Expansion of a union shorthand

Editorial note: Consider including tips on writing axioms from LP user’s guide.

10 User-defined data types

Users 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 14). This trait provides notations for two sorts (I and J) and five operators

\begin{verbatim}
first: I \rightarrow I
left, right: I \rightarrow I
name: I \rightarrow J
\end{verbatim}

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 LCM (Figure 16) 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(1, String) appearing in the definition of the automata Process (Figure 13) and LCR (Figure 15 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 18) 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 27), 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({}, mm ⊢ 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, m ⊢ [m, true]);
(subseqMarked(mms1, mms2) ∧ subseqMarked(mms2, mms3))
⇒ subseqMarked(mms1, mms3);

Figure 27: Definition of type constructor Mark[_.]

Redefining built-in type constructors The statement axioms Subsequence for Seq[_.] appearing before the definition of the automaton LossyChannel (Figure 8) overrides the built-in defi-
nition of the type constructor Seq[\_\_]. Ordinarily, axioms for that type constructor are obtained from a built-in trait Sequence[E]. 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, \( \leq \), and defines its properties.
Part III
IOA Reference Manual

An IOA specification contains four different kinds of units.
- Type definitions, used to represent state components or indices for automata (see Section 14).
- Automaton definitions (see Sections 13, 15, and 16).
- Assertions about automata, e.g., invariant and simulation relations (see Section 17).
- 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).

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 vocabulary \( \mathcal{V} \) for first-order logic is a set of symbols that denote two kinds of objects: sorts, denoted by symbols in \( \mathcal{V}_{\text{sorts}} \), and operators, denoted by symbols in \( \mathcal{V}_{\text{ops}} \). In IOA and LSL, symbols such as \( \text{Bool} \), \( \text{Set}[\text{Int}] \), and \( \text{T} \) denote sorts, and symbols such as \( 0 : \rightarrow \text{Int}, \text{+}, \text{*}, \text{Int} \rightarrow \text{Int}, \text{f:T} \rightarrow \text{T} \), and \( \_ \neq \_ : \text{S,S} \rightarrow \text{Bool} \) denote operators.

\( \mathcal{V}_{\text{sorts}}^* \) is the set of all finite sequences of elements of \( \mathcal{V}_{\text{sorts}} \), including the zero-length sequence. The set \( \mathcal{V}_{\text{signs}} \) of signatures for a vocabulary \( \mathcal{V} \) is the set of all pairs \( \langle \text{domain}, \text{range} \rangle \) in which \( \text{domain} \in \mathcal{V}_{\text{sorts}}^* \) and \( \text{range} \in \mathcal{V}_{\text{sorts}}^* \).

Associated with each operator, \( \text{op} \), in a vocabulary \( \mathcal{V} \) is an identifier, \( \text{op.id} \), and a signature, \( \text{op.sig} \), in \( \mathcal{V}_{\text{signs}} \). For example, in IOA and LSL, \( 0, +, * \), and \( \neq \) are operator identifiers and \( \rightarrow \text{Int}, \text{Int} \rightarrow \text{Int}, \text{T} \rightarrow \text{T} \), and \( \text{S,S} \rightarrow \text{Bool} \) are signatures (the sequence of sort symbols preceding the \( \rightarrow \) constitutes the domain, and the sort symbol following the \( \rightarrow \) 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, we restrict attention to vocabularies \( \mathcal{V} \) that contain the sort symbol \( \text{Bool} \), the 0-ary operators \( \text{true} \) and \( \text{false} \) with signature \( \rightarrow \text{Bool} \), the unary operator \( \neg \) with signature \( \text{Bool} \rightarrow \text{Bool} \), and the binary operators \( \land, \lor, \Rightarrow, \land \leftrightarrow \) with signature \( \text{Bool,Bool} \rightarrow \text{Bool} \). Furthermore, we

---

10 A logic in which \( \mathcal{V}_{\text{sorts}} \) contains more than one symbol is called multisorted.
generally restrict attention to vocabularies \( \mathcal{V} \) that contain, for every sort \( S \) in \( \mathcal{V}_{\text{sorts}} \), the binary operators \( = \) and \( \neq \) with signature \( S, S \rightarrow \text{Bool} \).

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 \( \mathcal{V}_{\text{sorts}} \). In IOA, symbols such as \( n: \text{Int} \) and \( x:\text{Set}[\text{Int}] \) are variables.

For any vocabulary \( \mathcal{V} \), a \( \mathcal{V} \)-term is an expression constructed, as described below, from the operators in \( \mathcal{V}_{\text{ops}} \) 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 \( \mathcal{V}_{\text{ops}} \) with signature \( T_1, \ldots, T_n \rightarrow T \) 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 \( \text{Bool} \) and any variable \( v \) over \( \mathcal{V} \), the expressions \( \forall v \ t \) and \( \exists v \ t \) are \( \mathcal{V} \)-terms. Their sort is \( \text{Bool} \). (The symbols \( \forall \) and \( \exists \) are quantifier symbols, and the term \( t \) is the scope of the quantifiers \( \forall v \) and \( \exists v \).)

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

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 \( \text{Bool} \). 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 concatenation of two strings, and a statement such as \( \forall x \forall y (x < y \Rightarrow \exists z (x < z \land z < y)) \) 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 \( [\cdot]_\mathcal{S} \) 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 \( T_1, \ldots, T_n \rightarrow T \) 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 \( \text{Bool} \), \( \text{true} \), \( \text{false} \), \( \neg \), \( \land \), \( \lor \), \( \rightarrow \), \( \leftrightarrow \), \( = \), or \( \neq \), as described in Section 11.1, we restrict our attention to \( \mathcal{V} \)-structures that interpret these symbols as in Figure 28.

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:

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

---

11 Logics that contain the operator \( = \) are called logics with equality. We do not consider logics without equality.
\[
[\text{Bool}]_s = \{\text{true}, \text{false}\}
\]
\[
[\text{true}]_s = \text{true}
\]
\[
[\text{false}]_s = \text{false}
\]
\[
[\neg]_s (x) = \text{true} \iff x = \text{false}
\]
\[
[\land]_s (x, y) = \text{true} \iff x = \text{true} \text{ and } y = \text{true}
\]
\[
[\lor]_s (x, y) = \text{true} \iff x = \text{true} \text{ or } y = \text{true}
\]
\[
[\implies]_s (x, y) = \text{true} \iff x = \text{false} \text{ or } y = \text{true}
\]
\[
[\iff]_s (x, y) = \text{true} \iff x = y
\]
\[
[\equiv]_s (x, y) = \text{true} \iff x \neq y
\]

Figure 28: Standard interpretation of boolean sort and logical operators

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

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

11.3 Further terminology

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

- \(S\) is a model of \(\phi\) iff \(\phi\) is true in \(S\), that is, iff \([\phi]_s = \text{true}\).
- \(T\) is consistent iff there is a \(\mathcal{V}\)-structure that is a model of every sentence in \(T\).
- \(\phi\) is a (logical) consequence of \(T\) iff \(\phi\) 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 \(\text{false} \notin T\).

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 \(S\) is clear from the context, we write \([\cdots]_s\) for \([\cdots]_s\).

12 Lexical syntax

We use the following conventions to describe the syntax of IOA (and also the syntax of LSL). Uppercase words and symbols enclosed in single quotation marks are terminal symbols in a BNF 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 | y | an x or a y
x? | an optional x
x* | zero or more x’s
x+ | one or more x’s
x, * and x; * | zero or more x’s, separated by commas or semicolons
x, + and 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`, `choose`, `components`, `const`, `do`, `eff`, `else`, `elseif`, `enumeration`, `fi`, `for`, `forward`, `from`, `hidden`, `if`, `in`, `input`, `internal`, `invariant`, `od`, `of`, `output`, `pre`, `signature`, `simulation`, `so`, `states`, `tasks`, `that`, `then`, `to`, `transitions`, `tuple`, `type`, `union`, `where`.
- Beginning comment character: `%`
- IDENTIFIERs 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 \texttt{\textbackslash bot} and \texttt{\textbackslash top}.
- OPERATORS: 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, \texttt{\textbackslash bot}, or \texttt{\textbackslash top}).
- Whitespace: `space`, `tab`, `newline`.
- Reserved for future use: `''`

13 Automaton definitions

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

Syntax of automaton definitions

```
specification ::= trait | ioaSpec
ioaSpec ::= (axioms | typedef | automatonDef | assertion)+
automatonDef ::= 'automaton' automatonName automatonFormals?
                      assumptions? (simpleBody | composition)
automatonName ::= IDENTIFIER
automatonFormals ::= '(' automatonFormal, + ')'
automatonFormal ::= IDENTIFIER, + ':=' (type | 'type')
assumptions ::= 'assumes' traitRef, +
```

The specification of a trait T is kept in a file named T.1sl. Each ioaSpec is kept in a file with a name of the form <filename>.ioa.
The syntax and semantics for the constructs mentioned here can be found in Section 14 (constructorDef, type, typeDef), Section 15 (simpleBody), Section 16 (composition), Section 17 (assertion), Section 19 (trait), and Section 23 (traitRef).

Static semantics

An automatonFormal that contains the keyword type denotes a sequence of formal types, each element of which is simple sort (cf. Section 20) 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, $\mathcal{V}_{spec}$, of an ioaSpec is the union of the vocabularies of its typeDefs and its axioms. The vocabulary, $\mathcal{V}_A$, of an automatonDef for an automaton named $A$ 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 closure, $cl(\mathcal{V})$, of a vocabulary $\mathcal{V}$ is $\mathcal{V}$ enriched by all built-in sorts, by all sorts obtained from the built-in sorts and sorts in $\mathcal{V}$ using sort constructors that are either built-in or defined by axioms, and by all operators on these sorts that are either built-in or defined by axioms.

- There can be at most one automatonDef for an automatonName in an ioaSpec.
- The automaton formals in each 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 must be in $cl(\mathcal{V})$. (This ensures that $cl(\mathcal{V}_A)$ satisfies the closure properties in Section 11.2.)

Logical semantics

The global theory of an ioaSpec is the union of the theories of its typeDefs and its axioms.

The local theory of an automatonDef is the union of the theories of the typeDefs in its assumptions with the global theory of the ioaSpec.

Editorial note: These definitions need to take account of the theory associated with $cl(\mathcal{V}_A)$, not just with $\mathcal{V}_A$.

- The global theory of an ioaSpec must be consistent.

14 Type and type constructor definitions

A type can be a primitive or a compound type. The syntax and semantics of each type is given by a built-in or user-supplied LSL trait (see Sections 9 and 19).

Syntax of type declarations

\[
\begin{align*}
\text{type} & : = \quad \text{simpleType} \mid \text{compoundType} \\
\text{simpleType} & : = \quad \text{IDENTIFIER} \\
\text{compoundType} & : = \quad \text{typeConstructor} \ ['[', \text{type}, '+'],'] \\
\text{typeConstructor} & : = \quad \text{IDENTIFIER}
\end{align*}
\]
typeDef ::= 'type' type '=' shorthand
axioms ::= 'axioms' axiomSet,+ 
axiomSet ::= traitRef
      | traitId 'for' typeConstructor '[' ',' ',',* ']'

The syntax and semantics for shorthand and traitRef can be found in Section 9.8 and 23.

Static semantics

With each type is associated a sort, namely, the sort that is lexically identical to the type.

The vocabulary of an axioms is the union of the vocabularies of its traitRefs. (The traits named by the traitIds it associates with typeConstructors do not contribute to this vocabulary.) The vocabulary of a typedef is the vocabulary of its shorthand.

- A type can be defined in at most one shorthand in an ioaSpec. Editorial note: Have front-end tool check 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 in the referenced trait.

Logical semantics

The theory of an axioms is the union of the theories of its traitRefs. (The traits named by the traitIds it associates with typeConstructors do not contribute to this theory.) The theory of a typedef is the theory of its shorthand.

15 Primitive automata

15.1 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 15.2, 15.3 and 15.4, respectively. The syntax and semantics of terms and predicates are given in Section 21.
Static semantics

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).

- 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 \( d(V_A) \).
- Each identifier in a term following the keyword const in an actionFormal, or in a predicate in a where must be an actionFormal in that action, in \( d(V_A) \), or a bound variable (cf. Section 11.1).
- The type of a term used as a const actionFormal cannot be type.

Logical semantics

- 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.

15.2 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

\[
\begin{align*}
\text{states} & : = \text{'states' state,}+ (\text{'so 'that' predicate})? \\
\text{state} & : = \text{IDENTIFIER ' :' type (':=' value)}? \\
\text{value} & : = \text{term | choice} \\
\text{choice} & : = \text{'choose' (variable 'where' predicate)}?
\end{align*}
\]

The syntax and semantics of predicate, term, and variable are given in Section 21.

Static semantics

- Each state variable (that is, each IDENTIFIER qualified by a type in a state) must be distinct from all other state variables and from all formal parameters of the automaton and its actions.
- The type of the initial value assigned to a state variable must be the same as the type of that variable.
• Each identifier in a term assigned as the initial value of a state variable must be a bound variable or in \(\mathcal{cl}(\mathcal{V}_A)\).

• Each identifier 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 the parameters and state variables of the automaton.

• Each identifier in the predicate restricting the initial values of the state variables is similarly limited, except that state variables can also appear in the predicate.

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

**Logical semantics**

• The set of start states, determined by the assignments and/or allowed by the predicates, must be nonempty. *Editorial note: Phrase formal semantics in terms of “For any model ...”.*

### 15.3 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**

\[
\begin{align*}
\text{transitions} & ::= \ '\text{transitions}' \ transition+ \\
\text{transition} & ::= \ \text{actionHead} \ \text{chooseFormals}? \ \text{precondition}? \ \text{effect}? \\
\text{actionHead} & ::= \ \text{actionType} \ \text{actionName} \ (\text{actionActuals} \ \text{where}?)? \\
\text{actionActuals} & ::= \ '(', \ \text{term},+, '?' )' \\
\text{chooseFormals} & ::= \ '\text{choose}' \ \text{varDcl},+ \\
\text{precondition} & ::= \ '\text{pre}' \ \text{predicate} \\
\text{effect} & ::= \ '\text{eff}' \ \text{program} \ ('\text{so}' \ 'that' \ \text{predicate})? \\
\text{program} & ::= \ \text{statement};+ \\
\text{statement} & ::= \ \text{assignment} | \ \text{conditional} | \ \text{loop} \\
\text{assignment} & ::= \ \text{lvalue} ':=' \ \text{value} \\
\text{lvalue} & ::= \ \text{variable} \\
& \quad | \ \text{lvalue} ' '[' \ \text{term},+ ' ]' \\
& \quad | \ \text{lvalue} '. .' \ \text{IDENTIFIER} \\
\text{conditional} & ::= \ 'if' \ \text{predicate} \ 'then' \ \text{program} \\
& \quad ( 'elseif' \ \text{predicate} \ 'then' \ \text{program})* \\
& \quad ( 'else' \ \text{program}? 'fi' \\
\text{loop} & ::= \ 'for' \ \text{IDENTIFIER} \ \text{qualification} \\
& \quad ( 'in' \ | \ 'so' \ 'that') \ \text{term} \ 'do' \ \text{program} 'od'
\end{align*}
\]

The syntax and semantics of predicate, qualification, variable, and term are given in Section 21.
Static semantics

- Transitions must be specified for all actionNames 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.
- The variables in the chooseFormals, if any, must be distinct from each other, from all automatonFormals, from all variables in the actionActuals of the action, and from all state variables.
- All operators, constants, and identifiers in a predicate in a precondition or conditional, or in a value or value in an assignment, must be
  - in d(V_A),
  - variables introduced in the actionActuals,
  - chooseFormals 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 identifiers in the predicate in a so that clause must satisfy the same restrictions or be primed state 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 d(V_A). The variable itself must be distinct from all variables in the automatonFormals, used as state variables, in the actionActuals, or in the chooseFormals.

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

  $$\exists h \ldots (pre(s) \land eff(s, s') \land soThat(s, s'))$$

  where

  - h \ldots are the choose formals, if any, in the transition,
- \( \text{pre}(s) \) is the predicate in the precondition,
- \( \text{eff}(s, s') \) is a formula obtained by translating the program, if any, in the effect, as described below, and
- soThat\( (s, s') \) is the predicate, if any, in the so that clause in the effect.

- The semantics of a program \( P \) is defined by translating it into a so that clause \( \text{eff}_P \), 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.

<table>
<thead>
<tr>
<th>program ( P )</th>
<th>( \text{eff}_P )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v := t )</td>
<td>( s'.v = t \land s'.w = s.w )</td>
</tr>
<tr>
<td>( P_1; P_2 )</td>
<td>( \exists s''(\text{eff}<em>{P_1}(s, s'') \land \text{eff}</em>{P_2}(s'', s')) )</td>
</tr>
<tr>
<td>if ( p \text{ then } P_1 \text{ fi} )</td>
<td>( (p \rightarrow \text{eff}_{P_1}(s, s')) \land (\neg p \rightarrow s' = s) )</td>
</tr>
<tr>
<td>if ( p \text{ then } P_1 \text{ else } P_2 \text{ fi} )</td>
<td>( (p \rightarrow \text{eff}<em>{P_1}(s, s')) \land (\neg p \rightarrow \text{eff}</em>{P_2}(s, s')) )</td>
</tr>
<tr>
<td>for ( v \text{ in } t \text{ do } P_1 \text{ od} )</td>
<td>( \forall x(v \in x \Rightarrow \text{eff}_{v:=x;P_1}(s, s')) )</td>
</tr>
</tbody>
</table>

- The formula \( \text{eff}(s, s') \) obtained by translating a program in an effect must be consistent.

- Identifiers for state variables in so that 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.

### 15.4 Automaton tasks

Tasks define a partition of the actions of an automaton.

**Syntax of tasks**

```
tasks ::= 'tasks' task;+
task ::= '{' actionSet '}'; forClause?
actionSet ::= actualAction,+ forClause?
actualAction ::= actionName actionActualls?
for ::= 'for' (IDENTIFIER ':' type),+ where?
```

**Static semantics**

- Each actionName in a task must be an internal or output action of the automaton.
- The number of actionActualls 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 \( \mathcal{d}(\mathcal{V}_A) \) 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.

16 Operations on automata

Automata can be constructed from previously defined automata by the operations of composition and hiding. Composite automata identify actions with the same name in different component automata; when any component automaton performs a step involving an action \( \pi \), so do all component automata that have \( \pi \) in their signatures. The hiding operator reclassifies output actions as internal actions.

Syntax of composite automata definitions

```plaintext
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 identifiers used as componentFormals must be distinct from each other and from any automatonFormal.

• The type of each componentFormal must be in \( \mathcal{d}(\mathcal{V}_A) \).

• Each automatonName must have been defined previously in an automatonDef.

• The numbers and types of the automatonActuals must match those of the corresponding automatonFormals.

• All identifiers in terms used as automatonActuals parameter must be in \( \mathcal{d}(\mathcal{V}_A) \), 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 product of the sets of states of the component automata.

• The set of start states of the composition is the product of the sets of start states of the component automata.

• A triple \((s, \pi, s')\) is in the transition relation for the composite automaton if, for every component automaton \(C\), \((s_C, \pi, s'_C)\) is a transition of \(C\) when \(\pi\) is an action of \(C\) and \(s_C = s'_C\) when \(\pi\) 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 preliminary description of these notations can be found at [nms.lcs.edu/~garland/IOA/stateVars.doc](http://nms.lcs.edu/~garland/IOA/stateVars.doc).

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.

17 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

\[
\text{assertion ::= invariant | simulation} \\
\text{invariant ::= 'invariant' 'of' automatonName ';'} \text{ predicate} \\
\text{simulation ::= ('forward' | 'backward') 'simulation' 'from' automatonName 'to' automatonName ';'} \text{ predicate}
\]

48
Static semantics

- Each automatonName must have been defined previously in an automatonDef.
- All operators, constants, and identifiers in a predicate in an assertion must be
  - in $d(\mathcal{V}_A)$ 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.

18 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.

19 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 23). The body of the trait consists of optional references to subtraits (Section 23) intermixed with shorthands defining sorts (Section 22), followed by sort and operator declarations (Section 20), axioms (Section 21), and claimed consequences of the axioms (Section 24).

Syntax of traits

\[
\text{trait} ::= \text{traitId} \text{traitFormals? '':' trait' traitBody} \\
\text{traitId} ::= \text{IDENTIFIER} \\
\text{traitBody} ::= (\text{subtrait} | \text{sort shorthand})^* \text{opDcls? axioms? consequences?}
\]

20 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. Operator names can be used in several different kinds of notations for terms.

<table>
<thead>
<tr>
<th>Operator declaration</th>
<th>Form of term</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>f: Int -&gt; Int</td>
<td>functional</td>
<td>f(i)</td>
</tr>
<tr>
<td>min: Int, Int -&gt; Int</td>
<td>&quot;</td>
<td>min(i, j)</td>
</tr>
<tr>
<td>0: -&gt; Int</td>
<td>&quot;</td>
<td>0</td>
</tr>
<tr>
<td><strong>&lt;</strong>: Int, Int -&gt; Bool</td>
<td>infix</td>
<td>i &lt; j</td>
</tr>
<tr>
<td>__&gt;: Int, Int -&gt; Int</td>
<td>prefix</td>
<td>-i</td>
</tr>
<tr>
<td>__!: Int, Int -&gt; Int</td>
<td>postfix</td>
<td>!</td>
</tr>
<tr>
<td>__.last: Seq[Int] -&gt; Int</td>
<td>&quot;</td>
<td>s.last</td>
</tr>
<tr>
<td><strong>[</strong>]: A, Int -&gt; V</td>
<td>bracketed</td>
<td>a[i]</td>
</tr>
<tr>
<td>{__}: E -&gt; Set[E]</td>
<td>&quot;</td>
<td>{x}</td>
</tr>
<tr>
<td>{}: -&gt; Set[E]</td>
<td>&quot;</td>
<td>{}</td>
</tr>
<tr>
<td>if__then__else__: Bool, S, S -&gt; S</td>
<td>conditional</td>
<td>if x &lt; 0 then \neg x else x</td>
</tr>
<tr>
<td></td>
<td>quantified</td>
<td>\forall x \exists y(x &lt; y)</td>
</tr>
</tbody>
</table>
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**

\[
\text{opDcls} ::= \text{'introduces'} \text{opDcl}^+ \\
\text{opDcl} ::= \text{name},+ \text{','} \text{signature },+,
\]

name ::= 'if', '_--', 'then', '_--', 'else', '_--'

| '_--'? \text{OPERATOR } '_--'? 

| '_--'? \text{openSym } '_--',* closeSym '_--'? 

| '_--'? ', IDENTIFIER 

| IDENTIFIER

openSym ::= '[' | '{' | '{' | '\(' | '<'

| '\langle' | '\lfloor' | '\lceil'

closeSym ::= ']' | '}' | '\)' | '>

| '\rangle' | '\rfloor' | '\rceil'

operator ::= \text{name ('::', signature)?}

signature ::= \text{domain '->' range}

domain ::= sort, *

range ::= sort

sort ::= simpleSort | compoundSort

simpleSort ::= IDENTIFIER

compoundSort ::= \text{sortConstructor '[', sort, +, ']'}

sortConstructor ::= IDENTIFIER


*Editorial note: Describe the parsing precedence for operators.*

**Static semantics**

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

- 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 \text{true} and \text{false} must be \text{->Bool}. Declarations for these operators are built into LSL.

- The signature of the logical operators \text{\&\&}, \text{\&}, \text{\&\&}, \text{\lor} must be \text{Bool,Bool -> Bool}. Declarations for these operators are built into LSL.

- The signature of the operators = and \not= must be \text{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 22).

- The signature of the operator if__then__else__ must be \text{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 22).
Logical semantics

- A sort denotes a non-empty set of elements.\textsuperscript{12}
- Different sorts denote disjoint sets of elements.
- An opDcl defines a list of \textit{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

21 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.

<table>
<thead>
<tr>
<th>Unparenthesized term</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x - y - z )</td>
<td>( (x - y) - z )</td>
</tr>
<tr>
<td>( a = b + c \Rightarrow b &lt; s(a) )</td>
<td>( (a = (b + c)) \Rightarrow (b &lt; s(a)) )</td>
</tr>
<tr>
<td>( a.b.c! )</td>
<td>((a.b).c! )</td>
</tr>
<tr>
<td>( \neg p \land \neg x.pre )</td>
<td>((\neg p) \land (\neg (x.pre)) )</td>
</tr>
<tr>
<td>( \exists x(x &lt; c) \Rightarrow c &gt; 0 )</td>
<td>((\exists x(x &lt; c)) \Rightarrow (c &gt; 0) )</td>
</tr>
<tr>
<td>( \forall x \exists y x &lt; y )</td>
<td>((\forall x \exists y) x &lt; y )</td>
</tr>
<tr>
<td>( a &lt; b + c )</td>
<td>Error</td>
</tr>
<tr>
<td>( p \land q \lor r )</td>
<td>Error</td>
</tr>
<tr>
<td>( p \Rightarrow q \Rightarrow r )</td>
<td>Error</td>
</tr>
</tbody>
</table>

Syntax of axioms

axioms ::= 'asserts' varDcls? axiom+ ';';?
varDcls ::= 'with' (IDENTIFIER,* qualification)+
qualification ::= '': sort
taxiom ::= predicate
     | 'sort' sort ('generated' 'freely'? | 'partitioned')
     | 'by' operator,+ predicate
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)?

\textsuperscript{12}LSL accords syntactic, but not semantic, meaning to compound sorts.
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 20), introduced by a shorthand for a sort (Section 22), or declared in a subtrait (Section 23).
- 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, OPERATORs, openSym, and closeSym 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.
- 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

- See Section 9.2 for the semantics of generated by and partitioned by axioms.

22 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

- See Section 9.8

23 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

\[
\begin{align*}
\text{subtrait} & ::= (\text{'includes'} \mid \text{'assumes'}) \text{traitRef},+ \\
\text{traitRef} & ::= \text{traitId} \text{renaming}\? \\
\text{traitId} & ::= \text{IDENTIFIER} \\
\text{renaming} & ::= \text{'}(\text{traitActual},+\text{'}) \\
& \mid \text{'}(\text{traitActual},\ast \text{replace},+\text{'}) \\
\text{replace} & ::= \text{traitActual} \text{FOR} \text{traitFormal} \\
\text{traitActual} & ::= \text{name} \mid \text{compoundSort} \\
\text{traitFormals} & ::= \text{'}(\text{traitFormal},\ast\text{'}) \\
\text{traitFormal} & ::= \text{name signature}\? \mid \text{compoundSort}
\end{align*}
\]

Static semantics

- There must not be a cycle in the assumes/includes hierarchy.
- 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 interpretation 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.

24 Consequences

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

Syntax of consequences

\[
\text{consequences ::= 'implies' varDcls? consequence;+ ',','}
\]
\[
\text{consequence ::= axiom | 'trait' traitRef,+ | conversion}
\]
\[
\text{conversion ::= 'converts' operator,+ ('excluding' term,+)?}
\]

Static semantics

• All sorts and operators in a consequence, including those declared in an implied traitFormal, 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 excluding 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 occurrence 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 occurrence of each $op$ in $Ops$ by $op'$.

25 Converts

*Editorial note:* Write this.
Part V
Appendices

A  Axioms for built-in data types

Editorial note: To be supplied.
B Software tools for IOA

IOA is being developed to enable the construction of a variety of software tools that support the description and analysis of concurrent algorithms. Among these tools will be the following:

- An LSL data type library, which will supply specifications of the data types built into IOA, as well as of other common abstract data types for use in describing I/O automata. The LSL Handbook [7], or a subset thereof, will form the basis for this library. Users will be able to extend the library.

- A library of LSL traits that provide precise definitions for the semantics of I/O automata and for relations between automata.

- A syntax and static semantic checker, for checking the well-formedness of descriptions for I/O automata.

- A prettyprinter, for tidying up descriptions of I/O automata.

- A simulator, for testing the behavior of I/O automata.

- Proof tools, to assist in the proof of invariants, simulation relations, and temporal properties. One such tool will be based on the Larch Prover [5]. Similar tools may be constructed for other verification systems, such as PVS [12], or for finite state model checkers, such as SMV [10] and SPIN [11].
C  Bibliography

References


