1 Excerpt From Event Systems Article

The following is an excerpt from a paper about the logic of events. All the formulas and terms in this paper were generated automatically by inserting references to objects in the FDL into the Latex source document. In the source document we do not even use the Latex math mode – all the math is generated from the FDL.

1.1 Event Systems

We want an abstract model that can capture the observable features of a distributed system. The fundamental types are *locations* and *events* which we can think of as space and time coordinates. Information is stored at a location as the value of a state variable or an *observable* and information is passed from one location to another along *links* in the form of *messages*.

Locations, observables (state variables), local action kinds, and message tags are all represented as members of the type `Id`, and links have type `IdLnk`. The types `Id`, and `IdLnk` are discrete types – equality on each type is decidable. The operations `source(l)` and `destination(l)` assign source and destination locations to the links, forming a graph structure on the locations and links.

A message will consist of a link, a tag, and a value whose type may depend on the link and the tag.

```
          Msg(M)  ≡  l:IdLnk × t:Id × (M l t)
      msg(l;t;v)  ≡  <l, t, v>
         mlnk(m)  ≡  m.1
         mtag(m)  ≡  m.2.1
         mval(m)  ≡  m.2.2
   haslink(l; m)  ≡  mlnk(m) = l
   Msg_sub(l; M)  ≡  {m:Msg(M)| haslink(l; m)}
    onlnk(l;mss)  ≡  filter(λms.mlnk(ms) = l;mss)
```

Every event will have a kind, a value, and a location. So an event is a point in spacetime. There are two kinds of events, local events have a kind that is an `Id` and the receipt of a message on a given link with a given tag is the other kind of event.

```
                                Knd  ≡  IdLnk × Id + Id
                          isrcv(k)   ≡  isl(k)
                        islocal(k)   ≡  ¬_b isl(k)
                        rcv(l; tg)   ≡  inl <l, tg>
                           locl(a)   ≡  inr a
                            lnk(k)   ≡  outl(k).1
                            tag(k)   ≡  outl(k).2
                            act(k)   ≡  outr(k)
kindcase(k;a.f[a];l,t.g[l; t])   ≡  if islocal(k)
                                    then f[act(k)]
                                    else g[lnk(k); tag(k)]
                                    fi
```

An *event system* is a structure consisting of a discrete type `E` of events and a set of operations and axioms. Operations `loc(e)`, `kind(e)`, and `val(e)` extract the location, kind, and value from an event. Operations `(x when e)` and `(x after e)` observe the values of the observables at the points in spacetime. Operation `first(e)` is a boolean that is true of only the first event at each location. Messages must originate at some point in spacetime, and the operations `sends(l;e)`,

`sender(e)`, and `index(e)` define this structure. The `sends(l;e)` of an event `e` on link `l` will be a list of messages on that link that originate at `e`. We build the semantics of message delivery into our model in a way that makes every link into a reliable fifo channel. Thus every message is eventually received, and for a receive event `e` the operations `sender(e)` and `index(e)` will provide the originator of the message received and the index of that message in the list that originated there. The temporal order structure on our spacetime is provided by two orderings on events `(e <loc e')` and `(e < e')`. The local ordering `(e <loc e')` is a total, discrete, well-founded, linear ordering on events with the same location. So, at each location, if there are any events, there must be a `<loc`-minimal event satisfying the predicate `first(e)`, and every non-minimal event `e` must have an immediate local predecessor `pred(e)`.

The causal ordering `(e < e')` is also well-founded and is the transitive closure of `(e <loc e')` and the relation that a receive event `e` is preceded by `sender(e)`.

Formally, an event systems is a member of the following dependent product type. This type is typical of the way structures are represented in type theory.

```
ES  ≡  E:𝕌
       × eq:EqDecider(E)
       × T:Id ⟶ Id ⟶ 𝕌
       × V:Id ⟶ Id ⟶ 𝕌
       × M:IdLnk ⟶ Id ⟶ 𝕌
       × unused:Top
       × loc:E ⟶ Id
       × kind:E ⟶ Knd
       × val:e:E ⟶ eventtype(kind;loc;V;M;e)
       × when:x:Id ⟶ e:E ⟶ (T (loc e) x)
       × after:x:Id ⟶ e:E ⟶ (T (loc e) x)
       × sends:l:IdLnk ⟶ E ⟶ (Msg_sub(l; M) List)
       × sender:{e:E| ↑isrcv(kind e)}  ⟶ E
       × index:e:{e:E| ↑isrcv(kind e)}  ⟶ ℕ||sends
                                          lnk(kind e)
                                          (sender e)||
       × first:E ⟶ 𝔹
       × pred:e':{e':E| ¬↑(first e')}  ⟶ E
       × causl:E ⟶ E ⟶ ℙ
       × p:ESAxioms(E;T;M;
                    loc;kind;val;
                    when;after;
                    sends;sender;index;
                    first;pred;
                    causl)
       × Top
```

The type `Top` at the end allows us to define subtypes of `ES` that have additional operators and axioms.

The axioms of the event system structure are the following

```
Trans(E;e,e'.(e <loc e'))                                          (1)
SWellFounded((e <loc e'))                                          (2)
∀e,e':E.                                                           (3)
  (loc(e) = loc(e')
  ⟺ (e <loc e') ∨ (e = e') ∨ (e' <loc e))
∀e:E. (↑first(e) ⟺ ∀e':E. (¬(e' <loc e)))                          (4)
```

```
∀e:E                                                                    (5)
  ((¬↑first(e))
  ⇒ ((pred(e) <loc e)
     ∧ (∀e':E
          (¬((pred(e) <loc e') ∧ (e' <loc e))))))
∀e:E                                                                    (6)
  ((¬↑first(e))
  ⇒ (∀x:Id. ((x when e) = (x after pred(e)))))
Trans(E;e,e'.(e < e'))                                                  (7)
SWellFounded((e < e'))                                                  (8)
∀e:E                                                                    (9)
  ((↑isrcv(e))
  ⇒ (sends(lnk(e);sender(e))[index(e)]
     = msg(lnk(e);tag(e);val(e))))
∀e,e':E.  ((e <loc e') ⇒ (e < e'))                                      (10)
∀e:E. ((↑isrcv(e)) ⇒ (sender(e) < e))                                   (11)
∀e,e':E.                                                                (12)
  ((e < e')
  ⇒ (((¬↑first(e'))
     c∧ ((e < pred(e')) ∨ (e = pred(e'))))
     ∨ ((↑isrcv(e'))
       c∧ ((e < sender(e')) ∨ (e = sender(e'))))))
∀e:E                                                                    (13)
  ((↑isrcv(e)) ⇒ (loc(e) = destination(lnk(e))))
∀e:E. ∀l:IdLnk.                                                         (14)
  ((¬(loc(e) = source(l))) ⇒ (sends(l;e) = []))
∀e,e':E.                                                                (15)
  ((↑isrcv(e))
  ⇒ (↑isrcv(e'))
  ⇒ (lnk(e) = lnk(e'))
  ⇒ ((e <loc e')
     ⟺ (sender(e) <loc sender(e'))
         ∨ ((sender(e) = sender(e'))
           ∧ (index(e) < index(e'))))))
∀e:E. ∀l:IdLnk. ∀n:ℕ||sends(l;e)||.                                     (16)
  ∃e':E
   ((↑isrcv(e'))
   c∧ ((lnk(e') = l)
      ∧ (sender(e') = e)
      ∧ (index(e') = n)))
```

1.1.1 Consequences of the axioms

We state as lemmas some properties that follow from the axioms.

```
∀e:E. (¬(e <loc e))                                                     (17)
∀e:E. (¬(e < e))                                                        (18)
∀e,e':E.                                                                (19)
  ((e <loc e')
  ⟺ (¬↑first(e')) ∧ ((e = pred(e')) ∨ (e <loc pred(e'))))
```

```
∀e,e':E.                                                              (20)
  (((e <loc e') ∧ (∀e1:E. (¬((e <loc e1) ∧ (e1 <loc e')))))
  ⇒ (e = pred(e')))
∀e',e:E.  Dec((e <loc e'))                                            (21)
∀e',e:E.  Dec((e < e'))                                               (22)
∀e:E. ∀l:IdLnk. ∀m:Msg.                                               (23)
  ((m ∈ sends(l;e))
  ⇒ ∃e':rvc(l,mtag(m),v).(e < e')
     ∧ ((msgtype(m) = valtype(e')) c∧ (v = mval(m))))
```

proofs: Lemmas 17 and 18 follow from the general fact that

$$WellFounded(Rel) \;\Rightarrow\; AntiReflexive(Rel)$$

Suppose `(e <loc e')`. From axiom (4) we conclude `¬↑first(e')`, and from axiom (5) we conclude

```
(pred(e') <loc e')
∧ (∀e'':E
(¬((pred(e') <loc e'') ∧ (e'' <loc e'))))
```

So `¬(pred(e') <loc e)` and hence, from axiom (3), `e ≤ pred(e')` , which proves lemma 19. If we also have

```
∀e1:E. (¬((e <loc e1) ∧ (e1 <loc e')))
```

then

```
¬(e <loc pred(e'))
```

so `e = pred(e')`, which proves lemma 20.

We may now prove lemma 21 by induction, using axiom (2). By lemma 19 it's enough to decide `(¬↑first(e')) ∧ e ≤ pred(e')` , but this is decidable by the induction hypothesis, and the decidability of equality in `E`. The proof of lemma 22 is similar. Using the other axioms we can show that axiom (12) can be proved as an if and only if statement, and hence it is enough to show that its righthand side is decidable. This follows from the induction hypothesis and the decidability of equality in `E`.

If `(msg(l;tg;v) ∈ sends(l;e))` then for some `n < ||sends(l;e)||`,

```
msg(l;tg;v) = sends(l;e)[n]
```

By axiom (16) there is an e' such that

```
((↑isrcv(e')) ∧ (lnk(e') = l)) ∧ (sender(e') = e) ∧ (index(e') = n)
```

So, by axiom (9),

```
(val(e') = mval(msg(l;tg;v)))
∧ (tg = mtag(msg(l;tg;v)))
```

That implies that `kind(e') = rcv(l; tg)` and since `e = sender(e')` we have `(e < e')` by axiom (11). This proves lemma 23.

1.1.2 Local histories

An event system is a rich enough structure that we can define various "history" operators that list or count previous events having certain properties. Because we can define operators like these we do not need to add "history variables" to the states in order to write specifications and and prove them.

The basic history operator lists all the prior events at a location.

Definition

```
before(e)   ≡  if first(e)
               then []
               else before(pred(e)) @ [pred(e)]
               fi
  [e, e']   ≡  filter(λev.es-ble{i:l}(es;e;ev);before(e') @ [e'])
```

```
 rcvs(l;before(e'))   ≡  filter(λe.haslnk(l;e);before(e'))
  snds(l;before(e))   ≡  concat(map(λe.sends(l;e);before(e)))
snds(l, before(e,n))  ≡  snds(l;before(e)) @ firstn(n;sends(l;e))
```

Using these operators we can state (and prove) the following important lemma.

Lemma Fifo

```
∀e:E
((↑isrcv(e))
⇒ (snds(lnk(e), before(sender(e),index(e)))
= msgs(lnk(e);before(e))))
```

proof: The proof is by induction on $<_{loc}$. The full proof is in then FDL.

□

1.1.3 Event system shorthands

We make some shorthand notations:

```
                ∀e@i.P[e]  ≡  ∀e:E. ((loc(e) = i) ⇒ P[e])
                ∃e@i.P[e]  ≡  ∃e:E. ((loc(e) = i) ∧ P[e])
          @i always.P[x]   ≡  ∀e@i.P[(x when e)]
     @i always.P[x1; x2]   ≡  ∀e@i.P[(x1 when e); (x2 when e)]
         ∃e=k(v).P[e; v]   ≡  ∃e:E. ((kind(e) = k) ∧ P[e; val(e)])
∃e:rvc(l,tg,v).P[e; v]     ≡  ∃e:E
                               ((↑isrcv(e))
                               ∧ (lnk(e) = l)
                               ∧ (tag(e) = tg)
                               ∧ P[e; val(e)])
```

1.2 Worlds

1.2.1 Definition of World

A *world* is a generalized trace of the execution of a distributed system. Time is modeled as the natural numbers $\mathbb{N}$. By observing the system at every location `i` and every time `t`, we have a state `s(i;t)`, an action `a(i;t)`, and a list of messages `m(i;t)`. The state `s(i;t)` is the state of the part of the system at location `i` at time `t`. We assume that the type of the state at location `i` does not change with time, and we use a general model of state as a record. A record is a dependent function. A world contains a type X of state variable names and and a type assignment $T : Loc \to X \to \mathbb{U}$. The state at location i of the world will have type $Record(X, T(i))$.

The action `a(i;t)` is the action that was chosen by the system to be executed next at location `i` and time `t`. It will always be possible that no action was taken at `i`,`t` so we must have a null action. Other action will be local actions with names taken from a type of action names A, and also the action of receiving a message. Every action will have a kind of one of these three forms (null, local, or receive), and also a value whose type depends on the kind and location of the action.

$$
\begin{array}{rcl}
\texttt{action(dec)} & \equiv & \texttt{Unit + (k:Knd} \times \texttt{(dec k))} \\
\texttt{isnull(a)} & \equiv & \texttt{isl(a)} \\
\texttt{kind(a)} & \equiv & \texttt{outr(a).1} \\
\texttt{val(a)} & \equiv & \texttt{outr(a).2} \\
\texttt{isrcv(l;a)} & \equiv & (\neg_b\texttt{isnull(a))} \\
 & & \wedge_b\ \texttt{isrcv(kind(a))} \\
 & & \wedge_b\ \texttt{lnk(kind(a)) = l}
\end{array}
$$

The messages `m(i;t)` are the list of messages sent from location `i` at time `t`. For messages, we use the message type `Msg(M)` defined earlier.

$$
\begin{array}{rcl}
\texttt{World} & \equiv & \texttt{T:Id} \to \texttt{Id} \to \mathbb{U} \\
 & & \times\ \texttt{TA:Id} \to \texttt{Id} \to \mathbb{U} \\
 & & \times\ \texttt{M:IdLnk} \to \texttt{Id} \to \mathbb{U} \\
 & & \times\ \texttt{s:i:Id} \to \mathbb{N} \to \texttt{x:Id} \to \texttt{(T i x)} \\
 & & \times\ \texttt{a:i:Id} \to \mathbb{N} \to \texttt{action(w-action-dec(TA;M;i))} \\
 & & \times\ \texttt{m:i:Id} \\
 & & \quad \to \mathbb{N} \\
 & & \quad \to \texttt{(\{m:Msg(M)| source(mlnk(m)) = i\} List)} \\
 & & \times\ \texttt{Top}
\end{array}
$$

If $w : World$ is a world, then we write w_{Loc}, w_{Lnk}, ..., w_s, w_a, and w_m for the components of w.

1.2.2 Fair-Fifo Worlds

We next define a *fair-fifo* world. We first note that, given world w, we can find all the messages sent on link l and all and receive actions that have occurred on link l before time t:

$$
\begin{array}{rcl}
\texttt{m(i;t)} & \equiv & \texttt{(w.2.2.2.2.2.1) i t} \\
\texttt{m(l;t)} & \equiv & \texttt{onlnk(l;m(source(l);t))} \\
\texttt{snds(l;t)} & \equiv & \texttt{concat(map(}\boldsymbol{\lambda}\texttt{t1.m(l;t1);upto(t)))} \\
\texttt{rcvs(l;t)} & \equiv & \texttt{filter(}\boldsymbol{\lambda}\texttt{a.isrcv(l;a);map(}\boldsymbol{\lambda}\texttt{t1.a(destination(l);t1);} \\
 & & \texttt{upto(t)))}
\end{array}
$$

The send and receive messages before time t define an implicit queue, and we can test whether the queue for link l is empty and for whether message ms is at the head of the queue for its link:

$$
\texttt{queue(l;t)} \equiv \texttt{nth\_tl(||rcvs(l;t)||;snds(l;t))}
$$

The predicate `FairFifo` is the conjuntion of the following four clauses

$$
\texttt{∀i:Id. ∀t:}\mathbb{N}\texttt{. ∀l:IdLnk.} \quad (24)
$$
$$
\texttt{((¬(source(l) = i)) ⇒ (onlnk(l;m(i;t)) = []))}
$$

$$
\texttt{∀i:Id. ∀t:}\mathbb{N}\texttt{.} \quad (25)
$$
$$
\texttt{((↑isnull(a(i;t)))}
$$
$$
\texttt{⇒ ((∀x:Id. (s(i;t + 1).x = s(i;t).x)) ∧ (m(i;t) = [])))}
$$

```
∀i:Id. ∀t:ℕ. ∀l:IdLnk.                                                    (26)
  ((↑isrcv(l;a(i;t)))
  ⇒ ((destination(l) = i)
     ∧ ((||queue(l;t)|| ≥ 1 )
       c∧ (hd(queue(l;t)) = msg(a(i;t))))))
∀l:IdLnk. ∀t:ℕ.                                                           (27)
  ∃t':ℕ
   ((t ≤ t')
   ∧ ((↑isrcv(l;a(destination(l);t')))
     ∨ (queue(l;t') = [])))
```

The first clause says that location i can only send message on links whose source is i. The second clause says that a null action leaves the state unchanged and sends no messages. The third clause says that a receive action at location i must be on a link whose destination is i and whose message is at the head of the queue. The fouth clause is the fairness clause. It says that for every queue, infinitely often either the queue is empty or a receive event occurs at its destination.

1.2.3 Event System of a World

If w is a fair-fifo world, then we can construct an event system from w. We have to define the type of events and define all the operations on events and show that the axioms are satisfied. Our events will be the points $\langle i, t\rangle$ in spacetime at which an action occured in w.

```
          E  ≡  {p:Id × ℕ| ¬↑isnull(a(p.1;p.2))}
     loc(e)  ≡  e.1
    time(e)  ≡  e.2
  Action(i)  ≡  action(w-action-dec(w.TA;w.M;i))
```

$$
\begin{aligned}
w_{state}(\langle i,t\rangle) &= w_s(i,t)\\
w_{state'}(\langle i,t\rangle) &= w_s(i,t+1)\\
w_{init}(i) &= w_s(i,0)\\
w_{msgs}(\langle i,t\rangle) &= w_m(i,t)
\end{aligned}
$$

For and event $e \in w_E$ we have $\neg isnull(w_{action}(e))$ so we may define

```
  kind(a)  ≡  outr(a).1
   val(a)  ≡  outr(a).2
```

The type of the value of an event can be determined from its location and kind using the type assignments w_{TA} and w_M as follows:

```
V(i;k)  ≡  kindcase(k;a.(w.2.1) i a;l,tg.(w.2.2.1) l tg)
```

The observation operators are defined in the obvious way:

```
  (x when e)  ≡  s(e.1;e.2).x
 (x after e)  ≡  s(e.1;(e.2) + 1).x
  sends(l;e)  ≡  onlnk(l;m(loc(e);time(e)))
```

The local ordering operations are also straightforward.

```
first(e)  ≡  if (time(e) =z 0) then tt
             if isnull(a(loc(e);time(e) - 1))
               then first(<loc(e), time(e) - 1>)
             else ff
             fi
```

```
pred(e)   ≡  if isnull(a(loc(e);time(e) - 1))
              then pred(<loc(e), time(e) - 1>)
              else <loc(e), time(e) - 1>
              fi
e <loc e' ≡  (loc(e) = loc(e')) ∧ (time(e) < time(e'))
```

To define the *sender* and *index* operations that match a receive event to its origin, we first define a *match* with the same *snds* and *rcvs* functions used in defining $FairFifo$.

```
match(l;t;t') ≡  ||snds(l;t)|| ≤z ||rcvs(l;t')||
                 ∧b ||rcvs(l;t')|| <z ||snds(l;t)||
                    + ||onlnk(l;m(source(l);t))||
```

Then, we define *sender* and *index* as follows

```
sender(e) ≡  <source(lnk(kind(e)))
             , mu(λt.match(lnk(kind(e));t;time(e)))
             >
index(e)  ≡  ||rcvs(lnk(kind(e));time(e))||
             - ||snds(lnk(kind(e));time(sender(e)))||
```

Finally, the causal ordering $\prec$ is defined as a transitive closure

```
e <c e' ≡  e
           λe,e'.
            (e <loc e'
            ∨ ((↑isrcv(kind(e'))) c∧ (e = sender(e'))))^+ e'
```

Putting all of these defined operations together, we have the event structure defined by the world and we can prove that the constructed event system satisfies all the event system axioms.
Theorem (World-Event-System)

```
∀the_w:World
(FairFifo
⇒ ESAxioms(E;λi,x.vartype(i;x);the_w.M;
λe.loc(e);λe.kind(e);λe.val(e);
λx,e.(x when e);λx,e.(x after e);
λl,e.sends(l;e);λe.sender(e);λe.index(e);
λe.first(e);λe.pred(e);
λe,e'.e <c e'))
```

proof: The full proof is in then FDL.

□

1.3 Message-Automata

Event systems and worlds are infinite objects, but they arise from the behaviors of distributed systems where, at each location, only a finite program constrains the behavior. We call our representations of these finite programs message-automata. To make our representations finite we need to replace infinite things like total type assignments with finite approximations, so we need some notation for finite partial functions.

1.3.1 Finite partial functions

The type `a:A fp-> B[a]` is the type of finite partial functions `f` from `A` to `B[a]` . Its domain is `dom(f)`, and we define

$$\texttt{f(x)?z} \equiv \texttt{if x} \in \texttt{dom(f) then f(x) else z fi}$$
$$\texttt{z != f(x) ==> P[a; z]} \equiv (\uparrow \texttt{x} \in \texttt{dom(f)}) \Rightarrow \texttt{P[x; f(x)]}$$

For finite partial functions `f` and `g` we define:

$$\begin{aligned}
\texttt{f} \subseteq \texttt{g} \equiv\ & \forall \texttt{x:A} \\
& ((\uparrow \texttt{x} \in \texttt{dom(f)}) \\
& \Rightarrow ((\uparrow \texttt{x} \in \texttt{dom(g)}) \ \texttt{c}\wedge\ (\texttt{f(x)} = \texttt{g(x)})))
\end{aligned}$$

$$\begin{aligned}
\texttt{f || g} \equiv\ & \forall \texttt{x:A} \\
& (((\uparrow \texttt{x} \in \texttt{dom(f)}) \wedge (\uparrow \texttt{x} \in \texttt{dom(g)})) \\
& \Rightarrow (\texttt{f(x)} = \texttt{g(x)}))
\end{aligned}$$

$$\begin{aligned}
\texttt{f} \oplus \texttt{g} \equiv\ & \texttt{<(f.1) @ filter(}\boldsymbol{\lambda}\texttt{a.(}\neg_b\texttt{a} \in \texttt{dom(f));g.1)} \\
& \texttt{, }\boldsymbol{\lambda}\texttt{a.f(a)?g(a)} \\
& \texttt{>}
\end{aligned}$$

Note that there is an empty partial function (with an empty domain) that is compatible with every finite partial function and is an identity operator with respect to `f` $\oplus$ `g`.

lemma

$\forall$`f,g:a:A fp-> B[a].` `(f || g` $\Rightarrow$ `{f` $\subseteq$ `f` $\oplus$ `g` $\wedge$ `g` $\subseteq$ `f` $\oplus$ `g})`

lemma

$\forall$`f,g:a:A fp-> B[a].` $\forall$`x:A.` $\forall$`P:a:A` $\rightarrow$ `B[a]` $\rightarrow$ $\mathbb{P}$.
`(g` $\subseteq$ `f` $\Rightarrow$ `z != f(x) ==> P[x;z]` $\Rightarrow$ `z != g(x) ==> P[x;z])`

1.3.2 Definition of Message-Automata

The message-automata share with the worlds and the event systems the same spaces of names for state variables, local action kinds, and message tags. But, where a world has, at each location i, type assignments $T(i) : X \to \mathbb{U}$, $TA(i) : A \to \mathbb{U}$, and $M : Lnk \to Tag \to \mathbb{U}$, a message-automaton will have finite type assignments (declarations)

$$ds : \texttt{x:Id fp-> } \mathbb{U}$$
$$da : \texttt{a:Knd fp-> } \mathbb{U}$$

The domain of ds is the set of declared state variables, the domain of da is the set of declared action kinds, both local actions and send/receive actions.

The state of a message-automaton will be the record defined by its declarations ds. We can define this as follows:

$$\texttt{State(ds)} \equiv \texttt{x:Id} \rightarrow \texttt{ds(x)?Top}$$

Here we extend the finite partial function ds to a total function by assigning the type Top to any undeclared state variable.

Every action has a value whose type depends only on the action kind. The type of the value associated with an action of kind k is defined by

$$\texttt{Valtype(da;k)} \equiv \texttt{da(k)?Top}$$

In addition, to its declarations, the message-automaton does the following things

init It constrains the initial values of the declared state variables. So, it has a finite partial function *init* of type `x:Id fp-> ds(x)?Void`. Thus, if x is in the domain of *init* then x is a declared state variable and $init(x)$ is a value of the declared type $ds(x)$ of state variable x.

pre It declares preconditions on its local actions. So, it has a finite partial function *pre* of type

$$\texttt{a:Id fp-> State(ds)} \longrightarrow \texttt{Valtype(da;locl(a))} \longrightarrow \mathbb{P}$$

Thus, if a is in the domain of *pre* then a is a declared local action and $pre(a)$ is a predicate on the state and the declared type $da(a)$ of the action.

ef It declares the effects of actions (local and input) on state variables. So, it has a finite partial function ef of type

$$\begin{array}{l}\texttt{kx:Knd} \times \texttt{Id fp-> State(ds)}\\ \longrightarrow \texttt{Valtype(da;kx.1)}\\ \longrightarrow \texttt{ds(kx.2)?Void}\end{array}$$

Thus, if $\langle k, x\rangle$ is in the domain of ef then k is an action kind and x is a declared state variable, and $ef(\langle k, x\rangle)$ is a function from the state and the action value to the type $ds(x)$ of x. This function defines how the new value of x will be computed from the current state and the value of the action.

send It declares the messages sent by actions. So, it has a finite partial function *send* of type

$$\begin{array}{l}\texttt{kl:Knd} \times \texttt{IdLnk fp-> (tg:Id}\\ \times \texttt{(State(ds)}\\ \longrightarrow \texttt{Valtype(da;kl.1)}\\ \longrightarrow \texttt{(da(rcv(kl.2; tg))?Void List))) List}\end{array}$$

Thus, if $\langle k, l\rangle$ is in the domain of *send* then k is an action kind and l is link. $snd(\langle k, l\rangle)$ is a list of pairs of type

$$\begin{array}{l}\texttt{tg:Id} \times \texttt{(State(ds)}\\ \longrightarrow \texttt{Valtype(da;kl.1)}\\ \longrightarrow \texttt{(da(rcv(kl.2; tg))?Void List))}\end{array}$$

For each pair $\langle tg, f\rangle$ in this list, the function f when applied to the current state and the value of the action gives a list of values of the type declared for link l and tag tg. The concatenation of all of these lists is the list of messages the action will send. This form for the send clause allows us to have conditional sends since the list returned by f may be empty. Also, a single action may send multiple messages on a link (and it may send on multiple links if there are other send clauses).

frame It declares implicit effects. By convention, the effects that are explicitly given are the only actions that affect the given state variables. So the implicit effect of any other action is to leave the state of variable unchanged. Since we want each clause of a message-automaton to be meaningful on its own, we can't depend on such contextual conventions, so we have to make the implicit effects explicit in so-called *frame* clauses. The message-automaton has a finite partial function $frame$ of type `x:Id fp-> Knd List`. So if x is in the domain of $frame$ then x is a declared state variable and $frame(x)$ is a list of actions kinds that contains all the kinds that affect x.

sframe It declares implicit sends. By convention, the sends that are explicitly given are the only actions that send messages on the given link with the given tag. So the implicit sends of any other action is to send no messages of the given link and tag. We make the implicit sends

explicit in *sframe* clauses. The message-automaton has a finite partial function $sframe$ of type `ltg:IdLnk × Id fp-> Knd List`. So if $\langle l, tg\rangle$ is in the domain of $sframe$ then l is an output link and $sframe(\langle l, tg\rangle)$ is a list of actions kinds that contains all the kinds that send messages with tag tg on link l.

Putting all of these pieces into a structure we define the type of message-automata:

```
MsgA  ≡  ds:x:Id fp-> 𝕌
         × da:a:Knd fp-> 𝕌
         × init:x:Id fp-> ds(x)?Void
         × pre:a:Id fp-> State(ds)
              ⟶ Valtype(da;locl(a))
              ⟶ ℙ
         × ef:kx:Knd × Id fp-> State(ds)
             ⟶ Valtype(da;kx.1)
             ⟶ ds(kx.2)?Void
         × send:kl:Knd × IdLnk fp-> (tg:Id
               × (State(ds)
                 ⟶ Valtype(da;kl.1)
                 ⟶ (da(rcv(kl.2; tg))?Void List))) List
         × frame:x:Id fp-> Knd List
         × sframe:ltg:IdLnk × Id fp-> Knd List
         × Top
```

Message-Automata `M1` and `M2` are compatible `M1 || M2` or satisfy the relation `M1` $\subseteq$ `M2` the eight finite partial functions, ds, da, $init$, pre, ef, snd, $frame$, and $sframe$ of `M1` and `M2` are compatible or are related by $\subseteq$. And we define `M1` $\oplus$ `M2` by applying the $\oplus$ operation to each of the components.

lemma

```
∀M1,M2:MsgA. M1 ⊆ M1 ⊕ M2
∀M1,M2:MsgA. (M1 || M2 ⇒ M2 ⊆ M1 ⊕ M2)
```

Note that there is an empty Message-Automaton in which every component is the empty partial function. The empty automaton is compatible with every automaton and is the identity wrt the join operation.

1.3.3 Distributed Systems

A distributed system is an assignment of a message automaton to every location. The message automaton at a location may be the empty automaton and the distributed system has finite support if the automaton at all but a finite number of locations is the empty automaton.

```
Dsys  ≡  i:Id ⟶ MsgA
```

We say that `D2` *extends* `D1` if

```
D1 ⊆ D2  ≡  ∀i:Id.  M(i) ⊆ M(i)
```

1.3.4 Semantics of Distributed Systems and Message-Automata

The semantics of a distributed system D is the set of possible worlds w that are consistent with it. To be consistent, w must have the same signature as D, be a fair-fifo world, and respect the meanings of the six components $init$, pre, ef, $send$, $frame$, and $sframe$ of the message-automata at

each location. The predicate `PossibleWorld(D;w)` is defined to be the conjunction of the following clauses

```
FairFifo                                                              (28)
∀i,x:Id.  (vartype(i;x) ⊆r M(i).ds(x))                                (29)
∀i:Id. ∀a:Action(i).                                                  (30)
  ((¬↑isnull(a))
  ⇒ (valtype(i;a) ⊆r M(i).da(kind(a))))
∀l:IdLnk. ∀tg:Id.                                                     (31)
  ((w.M l tg) ⊆r M(source(l)).da(rcv(l; tg)))
∀i,x:Id.  M(i).init(x,s(i;0).x)                                       (32)
∀i:Id. ∀t:ℕ.                                                          (33)
  ((¬↑isnull(a(i;t)))
  ⇒ (((↑islocal(kind(a(i;t))))
      ⇒ M(i).pre(act(kind(a(i;t))),λx.s(i;t).x,
         val(a(i;t))))
     ∧ (∀x:Id
          M(i).ef(kind(a(i;t)),x,λx.s(i;t).x,
          val(a(i;t)),s(i;t + 1).x))
     ∧ (∀l:IdLnk
          M(i).send(kind(a(i;t));l;λx.s(i;t).x;
          val(a(i;t));withlnk(l;m(i;t));i))
     ∧ (∀x:Id
          ((¬M(i).frame(kind(a(i;t)) affects x))
          ⇒ (s(i;t).x = s(i;t + 1).x)))
     ∧ (∀l:IdLnk. ∀tg:Id.
          ((¬M(i).sframe(kind(a(i;t)) sends
           <l,tg>))
          ⇒ (w-tagged(tg; onlnk(l;m(i;t)))
             = []))))))
∀i,a:Id. ∀t:ℕ.                                                        (34)
  ∃t':ℕ
   ((t ≤ t')
   ∧ (((¬↑isnull(a(i;t')))
      c∧ (kind(a(i;t')) = locl(a)))
     ∨ (¬a declared in M(i))
     ∨ unsolvable M(i).pre(a,λx.s(i;t').x)))
```

The following result is crucial to our theory:

Theorem

```
∀A,B:Dsys.
  (A ⊆ B
  ⇒ (∀w:World
        (PossibleWorld(B;w)
        ⇒ PossibleWorld(A;w))))
```

proof: For every $i \in Loc$, $M_1 = D_1(i) \subseteq M_2 = D_2(i)$. The definition of $PossibleWorld$ uses the automata $M \in \{M_1, M_2\}$ only in the context of conditional application of the finite partial functions, $M.init$, $M.pre$, $M.ef$, $M.send$, $M.frame$, and $M.sframe$, and also in some equality propositions over types $State(X, M.ds)$, $M.ds(x)$, and $List(Message(Lnk, Tag, M.dout))$. The

conditional applications all occur positively, and so the statement for M_2 implies the statement for M_1, by the definition of $M_1 \subseteq M_2$ and the lemma on conditional application of finite partial functions. The equalities also occur positively, and, so the equality for M_2 implies the equality for M_1 because $State(X, M_2.ds)$ is a subtype of $State(X, M_1.ds)$, and similarly, $M_2.ds(x)$ is a subtype of $M_1.ds(x)$ and $List(Message(Lnk, Tag, M_2.dout))$ is a subtype of $List(Message(Lnk, Tag, M_1.dout))$.