First-class Synchronous Operations

CML supports selective communication in a very general way, using first-class synchronous operations. The idea is to decouple the *description* of a synchronous operation from the *act* of synchronizing on it.

First-class Synchronous Operations

CML supports selective communication in a very general way, using first-class synchronous operations. The idea is to decouple the *description* of a synchronous operation from the *act* of synchronizing on it.

Synchronous operations are described as synchronization events, and may be synchronized on using the function `sync`:

```
type 'a event
val sync : 'a event -> 'a
```

Synchronizing on a value of type `'a event` will produce, if successful, a value of type `'a`. Synchronization may only succeed if an event is *enabled*.

Event Operations

```
(* base events *)
val sendEvt    : 'a chan * 'a -> unit event
val recvEvt    : 'a chan -> 'a event
val alwaysEvt  : 'a -> 'a event
val never      : 'a event
val timeOutEvt : Time.time -> unit event
val joinEvt    : thread_id -> unit event
(* combinators *)
val wrap   : 'a event * ('a -> 'b) -> 'b event
val choose : 'a event list -> 'a event
(* values provided for convenience/efficiency *)
val recv   : 'a chan -> 'a
val send   : 'a chan * 'a -> unit
val select : 'a event list -> 'a
recv   = sync o recvEvt
send   = sync o sendEvt
select = sync o choose
```

Example of Semantics

We'll study the semantics of CML later on. For now, though, let's look at an example. Suppose `ch1` is a `bool chan` and `ch2` is an `int chan`. Let ev be the `unit event`

```
choose
[wrap(sendEvt(ch1, true),
      fn () => print "send!\n"),
 wrap(recvEvt ch2,
      fn x => print("received: " ^ Int.toString x ^ "\n")),
 wrap(timeOutEvt(Time.fromSeconds 3),
      fn () => print "timeout!\n")]
```

Example of Semantics

We'll study the semantics of CML later on. For now, though, let's look at an example. Suppose `ch1` is a `bool chan` and `ch2` is an `int chan`. Let ev be the `unit event`

```
choose
[wrap(sendEvt(ch1, true),
      fn () => print "send!\n"),
 wrap(recvEvt ch2,
      fn x => print("received: " ^ Int.toString x ^ "\n")),
 wrap(timeOutEvt(Time.fromSeconds 3),
      fn () => print "timeout!\n")]
```

If we synchronize on ev, then CML will wait for at least one of the three base events to become enabled. At such points, ev itself is enabled, and CML may randomly choose one of the enabled events to synchronize on, supply the value of this synchronization to the function that is wrapped around it, and then return the value returned by this function (`()`, in all three cases) as the result of synchronizing on ev.

The Utility of First-class Synchronous Operations

The separation of synchronization into two parts allows:

- the definition of synchronization and communication abstractions that may be used in selective communication;
- selective communication involving dynamically computed possibilities.