First-class Synchronous Operations

John H. Reppy

AT&T Bell Laboratories

Abstract. The idea of making synchronous operations into first-class values is an important one for supporting abstraction and modularity in concurrent programs. This design principle has been used with great success in the concurrent language **CML**, but what are the limitations of this approach? This paper explains the rationale for first-class synchronous operations, and discusses their use in **CML**. It also presents some recent and fundamental results about the expressiveness of rendezvous primitives, which define the limitations of synchronous abstractions.

1 Introduction

Abstraction is a key tool for managing complexity. The design of programming languages is one area where application of this idea has paid significant dividends. Languages have evolved from providing a fixed set of abstractions of the underlying hardware, such as arithmetic expressions and arrays, to providing support for programmer-defined abstractions, such as abstract data-types and higher-order procedures. By providing mechanisms for the programmer to define her own abstractions, the language aids the management of application-specific complexity.

This paper looks one recent application of the principle that languages should support programmer-defined abstraction: *first-class synchronous operations.* First-class synchronous operations were developed originally as part of the **PML** language design [Rep88], and were further refined in the language *Concurrent ML* (**CML**) [Rep91, Rep95]. They provide a mechanism for programmers to define new synchronization and communication abstractions on top of a simple collection of primitives.

1.1 Concurrent programming

Concurrent programming is the task of writing programs consisting of multiple independent threads of control, called processes. Conceptually, we view these processes as executing in parallel, but in practice their execution may be interleaved on a single processor. For this reason, we distinguish between concurrency in a programming language, and parallelism in hardware. We say that operations in a program are *concurrent* if they could be executed in parallel, and we say that operations in hardware are *parallel* if they overlap in time.

One important motivation for concurrent programming is that processes are a useful abstraction mechanism: they provide encapsulation of state and control with well-defined interfaces. This paper focuses on the use of concurrent programming for applications with naturally concurrent structure. These applications share the property that flexibility in the scheduling of computation is required. Whereas sequential languages force a total order on computation, concurrent languages permit a partial order, which provides the needed flexibility.

The area of *interactive* systems is probably the richest source of examples of naturally concurrent systems. Interactive applications must deal with multiple asynchronous input streams and support multiple contexts, while maintaining responsiveness. Sequential implementations of interactive systems typically rely on the *faux* concurrency of event-loops and call-backs. A number of authors, ourselves included, have argued that programming interactive systems in a concurrent language results in cleaner and more flexible program structures [CP85, RG86, Pik89b, Pik89a, Haa90, GR92, GR93].

While interactive systems have been the main motivation for the author's work, concurrent program structures also appear in other application domains. Discrete event simulations, for example, can be structured as a collection of processes representing the objects being simulated. Distributed programs are, by their very nature, concurrent; each processor, or node, in a distributed system has its own state and control flow. In addition to the obvious concurrency that results from executing on multiple processors, there are natural uses of concurrency in the individual node programs.

1.2 Background

The ideas presented in this paper have been developed in the context of the **ML** family of languages, and we use the notation of *Standard ML* (**SML**) [MTH90] throughout. **SML** is a statically-typed higher-order language, which is used for a tasks ranging from theorem provers [Sli90] to low-level systems programming [BHLM94]. The interested reader is directed to either [Pau91] or [Ull94] for an introduction to the **SML** programming language.

Concurrent ML extends **SML** with support for dynamic thread creation and synchronous message passing. It is implemented as a library on top of the *Standard ML of New Jersey* system [AM91]; a discussion of **CML**'s implementation can be found in [Rep92]. The most significant use of **CML** is the multithreaded user interface toolkit **eXene** [GR93], and the applications built using **eXene** [Lin91, GM92, KSI94]. **CML** has also been used as the basis for research into distributed programming language design [Kru93].

1.3 Overview

In the next section, we motivate and introduce the concept of first-class synchronous operations in detail. Following this, we discuss **CML**, which is probably the most widely used language that provides first-class synchronous operations. To illustrate the usefulness of first-class synchronous operations, we describe a library for **CML** that supports distributed programming in the style of **Linda** [CG90]. We then examine the expressiveness of this approach, and show that there are inherent limitations on the amount of synchronization that one can implement as an abstraction, when starting from a particular set of primitives. Lastly, we discuss some other languages that provide mechanisms similar in nature to first-class synchronous operations, although they differ in purpose.

2 First-class synchronous operations

For the purposes of this discussion, we assume a traditional synchronous message-passing model with a *blocking* send operation. In an **SML** framework, this might have the following interface:

```
type 'a chan
val channel : unit -> '_a chan
val accept  : 'a chan -> 'a
val send    : ('a chan * 'a) -> unit
```

where the `channel` operation dynamically creates a new channel, and the `accept` and `send` operations implement the two halves of synchronous communication.

Most **CSP**-style languages (e.g., **occam** [Bur88] and **amber** [Car86]) provide similar rendezvous-style communication. In addition, they provide a mechanism for *selective communication*, which is necessary for threads to communicate with multiple partners. It is possible to use polling to implement selective communication, but to do so is awkward and requires busy waiting. Usually selective communication is provided as a multiplexed I/O operation. This can be a multiplexed input operation, such as **occam**'s `ALT` construct [Bur88]; or a generalized (or symmetric) select operation that multiplexes both input and output communications, such as **Pascal-m**'s `select` construct [AB86]. Implementing generalized select on a multi-processor can be difficult [Bor86], but as we discuss in Section 5, it is provably more expressive.

Unfortunately, there is a fundamental conflict between the desire for abstraction and the need for selective communication. For example, consider a server thread that provides a service via a *request-reply* (or *remote procedure call* (RPC) style) protocol. The server side of this protocol is something like:

```
fun serverLoop () = if serviceAvailable()
      then let
        val request = accept reqCh
        in
          send (replyCh, doit request);
          serverLoop ()
        end
      else doSomethingElse()
```

where the function `doit` actually implements the service. Note that the service is not always available. This protocol requires that clients obey the following two rules:

1. A client must send a request before trying to read a reply.
2. Following a request the client must read exactly one reply before issuing another request.

If all clients obey these rules, then we can guarantee that each request is answered with the correct reply, but if a client breaks one of these rules, then the requests and replies will be out of sync. An obvious way to improve the reliability of programs that use this service is to bundle the client-side protocol into a function that hides the details, thus ensuring that the rules are followed. The following code implements this abstraction:

```
fun clientCall x = (send(reqCh, x); accept replyCh)
```

While this insures that clients obey the protocol, it hides too much. If a client blocks on a call to `clientCall` (e.g., if the server is not available), then it cannot respond to other communications. Avoiding this situation requires using selective communication, but the client cannot do this because the function abstraction hides the synchronous aspect of the protocol. This is the fundamental conflict between selective communication and the existing forms of abstraction. If we make the operation abstract, we lose the flexibility of selective communication; but if we expose the protocol to allow selective communication, we lose the safety and ease of maintenance provided by abstraction.

To resolve the conflict between abstraction and selective communication requires introducing a new abstraction mechanism that preserves the synchronous nature of the abstraction. First-class synchronous operations provide this abstraction mechanism [Rep88, Rep91]. The basic idea is to introduce a new type constructor, `event`, for typing synchronous operations, much the same way that we introduce the `->` type constructor to type functional values. This allows us to represent synchronous operations as first-class values, instead of merely as functions, which, in turn, allows us to define a collection of combinators for defining new event values from some small collection of primitives. In this scheme, selective communication is expressed as a choice among event values, which means that user-defined abstractions can be used in a selective communication without breaking the abstraction.

The choice of combinators is motivated by analyzing the traditional `select` construct, which has four facets: the individual I/O operations, the actions associated with each operation, the nondeterministic choice, and the synchronization. To make this concrete, consider the following loop (using an **Amber** style `select` construct [Car86]), which implements the body of an accumulator that accepts either addition or subtraction input commands and offers its contents:

```
fun accum sum = (
      select addCh?x => accum(sum+x)
          or subCh?x => accum(sum-x)
          or readCh!sum => accum sum)
```

The `select` construct consists of three I/O operations: `addCh?x`, `subCh?x`, and `readCh!sum`. For each of these operations there is an associated action on the right hand side of the =>. Taken together, each I/O operation and associated action define a clause in a nondeterministic synchronous choice. It is also worth noting that the input clauses define a scope: the input operation binds an identifier to the incoming message, which has the action as its scope.

In the framework of first-class synchronous operations, each of these facets is supported independently. We start with *base-event* values to represent the communication operations, and provide combinators to associate actions with events and to build nondeterministic choices of events. Lastly, an operation is provided to synchronize on an event value. This decomposition of the select construct results in a flexible mechanism for building new synchronization and communication abstractions. We postpone discussion of the details of these until the next section.

One of the benefits of this framework is that other synchronous operations can be added uniformly by introducing new primitive event constructors. These can include timeout events, blocking input/output operations, and even other synchronization mechanisms, such as synchronizing memory operations [ANP89, BNA91]. Because these additional primitives are just event-value constructors, they can be combined freely. For example, adding a timeout to a message-passing operation or to a blocking I/O operation is easily done using the choice combinator and the timeout constructor. This should be contrasted with **Ada**, for example, where there are two different syntactic forms provided to support timeouts [DoD83].

3 Concurrent ML

Concurrent ML is *higher-order* concurrent language. By this we mean that it supports sequential higher-order programming in the form of first-class function values, and concurrent higher-order programming in the form of first-class synchronous operations. In this section, we give a brief description of the core **CML** features, and some examples of their use.

Figure 1 gives the signature of the most commonly used **CML** primitives. The first four of these operations are fairly standard:

`spawn` *f* creates a new thread to evaluate the function *f*, and returns the ID of the newly created thread.

`channel ()` creates a new channel. The "_" in its return type denotes an *imperative* type variable, which is a technical restriction necessary to ensure type soundness [Tof90, Rep92].

`accept` *ch* accepts a message from the channel *ch*. This operation blocks the calling thread until there is another thread sending a message on *ch*.

`send (`*ch*`,` *msg*`)` sends the message *msg* on the channel *ch*. This operation blocks the calling thread until there is another thread accepting a message from *ch*.

The remaining functions support the mechanism of first-class synchronous operations (or events):

```
type thread_id
type 'a chan
type 'a event

val spawn : (unit -> unit) -> thread_id

val channel  : unit -> '_a chan
val accept   : 'a chan -> 'a
val send     : ('a chan * 'a) -> unit

val receive  : 'a chan -> 'a event
val transmit : ('a chan * 'a) -> unit event

val guard    : (unit -> 'a event) -> 'a event
val wrap     : ('a event * ('a -> 'b) -> 'b event
val choose   : 'a event list -> 'a event

val sync     : 'a event -> 'a
val select   : 'a event list -> 'a
```

Fig. 1. The common **CML** operations

`receive` *ch* constructs an event value that represents the operation of accepting a message from the channel *ch*. When this event is synchronized on, it accepts a message from the channel and returns it as the synchronization result.

`transmit (`*ch*`,` *msg*`)` constructs an event value that represents the operation of sending the message *msg* on the channel *ch*. When this event is synchronized on, it sends the message on the channel and returns unit as the synchronization result.

`guard` *f* creates an event out of the pre-synchronization action *f*, which is called a *guard function*. When this value is synchronized on, the guard function is evaluated and its result is used in the synchronization.

`wrap (`*ev*`,` *f*`)` wraps the event value *ev* with the post-synchronization action *f*, which is called a *wrapper function*. When this event is synchronized on, the function *f* is applied to the synchronization result of *ev*.

`choose [`ev_1, ..., ev_n`]` forms the nondeterministic choice of the event values ev_1, ..., ev_n. When this event is synchronized on, one of the events ev_i will be selected, and its result will be the synchronization result.

`sync` *ev* forces synchronization of the event value *ev*, returning the synchronization result.

`select [`ev_1, ..., ev_n`]` synchronizes on the nondeterministic choice of the event values ev_1, ..., ev_n. This is the composition of `sync` and `choose`, and is provided for notational convenience.

We illustrate these operations with some examples, a formal semantics can be found in either [Rep92] or [Rep95].

The simplest example is the definition of the synchronous message passing operations:

```
fun accept ch = sync (receive ch)
fun send (ch, msg) = sync (transmit (ch, msg))
```

Another example is the example of selective communication given in the previous section. In **CML**, this would be written as:

```
fun accum sum = select [
        wrap (receive addCh, fn x => accum(sum+x)),
        wrap (receive subCh, fn x => accum(sum-x),
        wrap (transmit(readCh, sum), fn () => accum sum)
      ]
```

Here we are using `receive` instead of `?`, `transmit` instead of `!`, and `wrap` for associating actions with the communications.

3.1 A new abstraction — swap channels

The purpose of first-class synchronous operations is to provide a mechanism for defining new synchronization and communication abstractions. In this section, we present a simple example of a communication abstraction called a *swap channel*. This is a new type of channel that allows two processes to swap values when they rendezvous (one might call this *symmetric rendezvous*, since there is no sender or receiver distinction). The interface consists of an abstract type:

```
type 'a swap_chan
```

and functions for creating channels and "*swap*" events:

```
val swapChannel : unit -> '_a swap_chan
val swap        : ('_a swap_chan * '_a) -> '_a CML.event
```

When two processes communicate on a swap channel, each sends a value and each receives a value. Implementing this correctly requires some care.

Our implementation is based on the asymmetric message passing operations, so to insure the symmetry of the operation, each thread in a swap must both offer to send a message and to accept a message on the same channel. We can use the `choose` combinator for this purpose. Once one thread completes a send (which means that the other has completed an accept), we need to complete the swap, which means sending a value in the other direction. A first solution might be to use another channel for completing the swap, but unfortunately, this results in a race condition when two separate pairs of threads attempt swap operations on the same swap channel at the same time. For example, Figure 2 shows what might happen if threads P_1, P_2, Q_1 and Q_2 all attempt a swap operation on the same swap channel at roughly the same time (where the swap channel is represented by the channels `ch` and `ch'`). In this scenario, P_1 and P_2 are initially paired, as are Q_1 and Q_2. But there is a race in the second phase of the swaps, and P_1 ends up being paired with Q_2, and P_2 ends up with Q_1

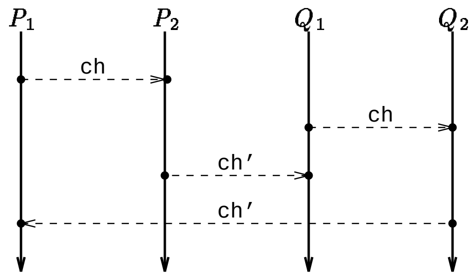

Fig. 2. A swap mismatch

```
datatype 'a swap_chan = SC of ('a * 'a chan) chan

fun swapChannel () = SC(channel ())

fun swap (SC ch, msgOut) = guard (fn () => let
      val inCh = channel ()
      in
        choose [
            wrap (receive ch,
              fn (msgIn, outCh) => (send(outCh, msgOut); msgIn)),
            wrap (transmit (ch, (msgOut, inCh)),
              fn () => accept inCh)
          ]
      end)
```

Fig. 3. The swap channel implementation

(because Q_1 beats P_1 to reading a value from `ch'`, and P_2 beats Q_2 to sending a value on `ch'`).

To avoid this problem, we need to allocate a fresh channel for completing the second phase of the swap operation each time the swap operation is executed. Figure 3 gives the implementation of the swap channel abstraction. The representation of a swap channel is a single channel that carries both value communicated in the first phase, and a channel for communicating the other value in the second phase. The channel for the second phase must be allocated before the synchronization on the first message, which is the kind of situation for which `guard` was designed. The guard function first allocates the channel for the second phase, and then constructs the swap event using the `choose` combinator. Because of the symmetry of the operation, each side of the swap must allocate a channel, even though only one of them will be used. Fortunately, allocating a channel is very cheap, and the garbage collector takes care of reclaiming it after we are done.

The swap channel example illustrates several important **CML** techniques. The use of dynamically allocating a new channel to serve as a unique identifier for a transaction comes up quite often in client-server protocols. It is also an example where the nondeterministic choice is used inside an abstraction, which is why we have both `choose` and `sync`, instead of just the `select` operator. And, lastly, this example illustrates the use of the `guard` combinator.

3.2 Negative acknowledgements

One place where **CML** extends the traditional synchronous message-passing operations is in the support for *negative acknowledgements*. When implementing client-server protocols, it is sometimes necessary for the server to know whether to *commit* or *abort* the transaction. The synchronization involved in sending the reply allows the server to know when the client has committed to the transaction, but there is no way to know when the client has aborted the transaction by selecting some other event. While a timeout on the reply might work in some situations, the absence of a reply does not guarantee that the client has selected some other event; the client just might be slow. To insure the correct semantics, we need a mechanism for *negative acknowledgements*. Furthermore, we would like this mechanism to be compatible with the abstraction provided by event values. To achieve this, **CML** provides the event combinator:

```
val withNack : (unit event -> 'a event) -> 'a event
```

which is a variation of the `guard` combinator.[1] As does `guard`, `withNack` takes a function whose evaluation is delayed until synchronization time, and returns an event value. The main difference is that at synchronization time, the guard function is applied to an *abort event*, which is enabled only if some other event is selected in the synchronization.

To make the workings of negative acknowledgements more concrete, consider the following simple use of `withNack`:

```
fun mkEvent (ev, ackMsg, nackMsg) = withNack (fn nack => (
      spawn (fn () => (sync nack; CIO.print nackMsg));
      wrap(ev, fn _ => CIO.print ackMsg))
```

This function takes an event, and returns one that will print the message `ackMsg`, if it is chosen, and the message `nackMsg` if another event is chosen.[2] Printing the acknowledgement is easy — `mkEvent` wraps the event with a wrapper function that prints `ackMsg`. To print the negative acknowledgement message, `mkEvent` spawns a thread that waits for the negative acknowledgement event `nack` and then prints `nackMsg`. If the event is chosen, then the `nack` event will never be

[1] Earlier versions of **CML** used a different combinator for supporting abort actions, called `wrapAbort` [Rep91, Rep92]. The two combinators have the same expressive power, but the `withNack` combinator is more natural to use. See [Rep95] for a discussion of this.

[2] The function `CIO.print` prints a string to the standard output.

enabled, and thus the nackMsg will not be printed (instead, the thread will be garbage collected). For example, consider what happens ch2 is chosen when executing the following synchronization:

```
select [
    mkEvent (receive ch1, "ch1\n", "not ch1\n"),
    mkEvent (receive ch2, "ch2\n", "not ch2\n")
  ]
```

This will cause the messages "ch2" and "not ch1" to be printed (the actual order of the messages is nondeterministic).

As a more realistic example, consider the implementation of a *lock service*. This service provides a collection of locks that can be acquired or released (much like mutex locks). The interface is:

```
type lock

val acquireLock : lock -> unit event
val releaseLock : lock -> unit
```

where the type lock is an abstract lock identifier. For purposes of this example, we ignore the question of how lock names are created.

If a client attempts to acquire a lock that is held, then it must wait until the holder of the lock releases it. Since a client may not want to wait indefinitely, the acquireLock operation is event-valued so that it can be used in selective communication (with a timeout, for example). This is clearly a case where we want to synchronize on the acquiring of the lock, rather than on the sending of the request. Thus, we need to use a guard function to send the request, and return an event that represents the acquisition of the lock.

The server maintains a set of locks that are currently held; for each lock in the set, it keeps a queue of pending requests for the lock. When a lock is released, the next requesting thread on the queue gets the lock. There is a subtlety to the servicing of pending requests. Consider the case where a client A requests a lock l that is held by some other thread B. By the time B releases l, A may no longer be interested in it (e.g., its attempt to synchronize on acquiring the lock may have timed out). In this case, if the server still thinks that A wants the lock l, it will assign l to A, resulting in starvation for any future attempt to acquire the lock. Clearly, this is a problem — we need to structure the protocol so that the server gets the *negative* information that requesting client is no longer interested in the lock.

While the details of the implementation are beyond the scope of this paper, the basic idea is simple. To do this, we implement the client side of the acquireLock protocol using the withNack combinator wrapped around a guard function that constructs and sends the acquire request to the server. The request consists of a freshly allocated channel for the server to use to notify the client that the lock is acquired, and the abort event that was passed to the guard function. The client synchronizes on receiving the notification message from the server, while the server synchronizes on the choice of sending the notification and the abort event. The interested reader is directed to [Rep95] for the complete story.

4 Extending CML for distributed programming

One of the principle uses of concurrency is for distributed programming. A distributed program consists of processes running in different address spaces on logically different processors. Because the processes are physically disjoint, there are a number of issues that come up in distributed systems that are not present in concurrent programming:

- communication latency is significantly higher over a network (e.g., on the order of several milliseconds on a high-performance workstation, versus less than 50μS for a **CML** `send/accept` on a SPARCstation-2).
- processors and network links can go down, and come back up, during the execution of a distributed program.
- because programs running on different nodes may be compiled independently, type security cannot be guaranteed statically.

For these reasons, the synchronous model used in **CML** does not map well to the distributed setting. One needs to provide a different programming model for dealing with remote communication.

Although they may not be directly usable for distributed programming, concurrent programming languages, such as **CML**, do have a role in the distributed setting. Since the individual programs running on the different processors must manage communication with each other, it is convenient to structure these programs as concurrent programs. For this reason, most distributed programming languages and toolkits provide concurrent programming features as well.

In this section, we examine an implementation of one possible model for distributed programming, called *tuple spaces*. This model has been popularized by the **Linda** family of programming languages. The implementation discussed in this section illustrates both how **CML** can be used to implement low-level systems services, and it shows how the **CML** primitives can provide a useful interface to asynchronous distributed programming primitives.[3]

4.1 Tuple spaces

Tuple spaces are a model of communication for parallel distributed-memory programming [GB82, Gel85]. A tuple space is basically an associative memory that is shared by the processes of a system. Unlike hardware memory, the values in a tuple space are tuples of typed fields, which are similar to **SML** tuples. Processes communicate by reading and writing tuple values. Reading a tuple is done by *pattern matching* on a subset of the tuple's fields.

Tuple spaces have been popularized by **Linda**, which is a family of programming languages. The **Linda** languages are traditional sequential languages, such as **C**, extended with a small collection of tuple-space operations. There are four basic operations in the **Linda** model:

[3] This example was inspired by Cooper and Siegel's **SML** implementation of **Linda** [SC91].

out is used to add a tuple to the tuple space.

in takes a tuple *template* (or *pattern*) as an argument, and removes a tuple that matches the template from the tuple space, binding the values in the tuple to the formal parameters in the template. If there is no tuple that matches the template, then the process suspends until the **in** can complete.

rd is like **in**, but does not remove the tuple from the tuple space.

eval is used to add an *active* tuple to the tuple space. This is the mechanism for creating new processes. A new process is created that computes the tuple value; when it finishes, the tuple becomes *passive*, and may be read by other processes.

There are also non-blocking variants of **in** and **rd**, called **inp** and **rdp**.

Communication in **Linda** differs from most other distributed-memory communication models, in that the messages (i.e., tuples) are not directed at any particular destination. Instead, once inserted into tuple space, a tuple can be read by any process that executes a matching **rd** operation. This flexibility of communication is further aided by the fact that both the templates used in **rd** operations, and the tuples used in **out** operations can have undefined fields, called *formals*. A formal matches any field of the right type, no matter what its value. When used in a template, a formal also provides a mechanism to bind the values in the tuple being read to local variables in the process.

4.2 CML-Linda

In this section, we describe the **CML** interface to **Linda**-style primitives. In their full generality, tuple-spaces are quite hard to implement with any sort of efficiency. For this reason, most implementations of tuple spaces constrain the model in various ways to make the implementation more tractable. The most common restriction is the requirement that the first element of every tuple be a defined string-valued field, which serves as the primary key for tuple matching. Another common restriction is only allowing formals in patterns, and not in the tuples in tuple-space. We make both of these restrictions in **CML-Linda**; furthermore, we do not implement the **eval** operation.

Figure 4 gives the signature of the **CML-Linda** interface. We use **SML** datatypes to represent the fields of tuples and templates. A tuple consists of an initial string-valued field followed by a list of zero or more values, while a template is an initial string-valued field followed by a list of pattern fields, where a pattern field is either a typed formal, an untyped wildcard, a tuple of patterns, or a constant value. The two blocking input operations (`input` and `read`) are provided as event-valued constructors to allow clients to use them in selective communications (e.g., in conjunction with timeouts). This illustrates how first-class synchronous operations can play a role in managing asynchronous distributed communication.

```
datatype value
  = INT of int
  | STR of string
  | BOOL of bool
  | TUPLE of value list

datatype pat
  = INT_FORMAL
  | STR_FORMAL
  | BOOL_FORMAL
  | WILDCARD
  | CONST of value
  | TUPLEPAT of pat list

type tuple_space

val output : tuple_space -> (string * value list) -> unit
val input  : tuple_space -> (string * pat list) -> value list event
val read   : tuple_space -> (string * pat list) -> value list event
```

Fig. 4. CML-Linda interface

4.3 Implementation issues

While space does not permit a detailed description of the implementation, we give an overview with some details (a full description can be found in [Rep95]).

The main design issue is how to distribute the tuples of a tuple space over the processors in a system. There are two basic ways to implement a distributed tuple space [BCGL87]:

- *Read-all, write-one:* where the tuples are distributed over the processors. A write operation adds a tuple to some processor, while a read operation must query all processors for a match.
- *Read-one, Write-all:* where each processor has a copy of every tuple. A write operation must add the tuple to all processors, while a read operation need only query the local processor's copy of tuple space.

For our implementation, we use the first technique.

With a read-all, write-one distribution, the implementation of the `write` operation is straightforward. When a tuple is written, some decision must be made as to where the tuple should be placed, and then a message is sent to the appropriate processor. Simple policies for tuple placement include always placing tuples locally, and round-robin placement. More sophisticated schemes might use static analysis or dynamic information for better placement.

The input operations (`input` and `read`) require more complicated protocols. The basic protocol for `input`, which is adopted from [SC91], is as follows:

1. The reading processor, call it P, broadcasts the template to all the tuple-space servers in the system.
2. Each tuple-space server checks for a match; if it finds one, then it places a *hold* on the tuple and sends the tuple to P, otherwise, it remembers the read request. If some subsequent `write` matches the read request, then it will then send the tuple to P.
3. The processor P waits for a matching tuple. Once P receives a matching tuple, say from Q, it sends Q a message acknowledging the tuple, and sends a *cancellation* message to the other processors.
4. When Q receives the acknowledgement, it removes the tuple from its space. When a processor that has a matching tuple receives a cancellation message, it cancels the hold it placed on the tuple, which allows the tuple to be used to match other requests. When a processor that does not have a matching tuple receives a cancellation message, it removes the record of the original request.

The protocol for `read` is similar; the main difference is that the tuple is not removed in step 4.

For each `input` (or `read`) request, we spawn a new thread to manage the operation. This thread broadcasts the template to be matched to the tuple servers, and coordinates the replies. Once a matching tuple has been found, it is communicated to the reading thread, and the cancellation messages are sent out to the other processors. Since the `input` and `read` operations are event valued, there is the added complication that these operations can be used in a selection context, where they must be aborted. We use negative acknowledgements to handle this situation. The basic implementation of the `input` operation has the form:

```
fun input ts req = withNack (fn nackEvt => let
      val replyCh = channel()
      in
        spawn (inputManager (ts, replyCh, nackEvt, req));
        receive replyCh
      end)
```

Of course, most of the details are hidden in the implementation of the manager thread. In addition to waiting for a matching tuple, the manager thread also waits for the `nackEvt` to be enabled. If the manager thread receives a negative acknowledgement, then it sends cancellation messages to all of the tuple-space servers. We also take care not to send the acknowledgement message until we are sure that the tuple has been accepted. We do this by synchronizing on the choice of sending the matching tuple on the `replyCh` and receiving the negative acknowledgement. The semantics of **CML** guarantee that only one of these can occur.

The `input` and `read` operations are good examples of how complex protocols can be abstracted as simple synchronous operations.

5 Limitations of CML events

When proposing a new abstraction mechanism, it is fair to ask "what are the limitations?" What things can be expressed in a non-abstract way that cannot be expressed abstractly. For **CML**events, this question can be stated as:

> For a given set of base-event constructors, and combinators, what kinds of synchronization abstractions can we build?

By *abstract*, we mean independent of context, which in this case means can be placed in a selective communication. Recent work by Prakash Panangaden and the author have explored this question using a knowledge-theoretic framework [PR94]. The main result of this work is the following theorem:

Theorem. *Given the standard* **CML** *event combinators and an n-way rendezvous base-event constructor, one* cannot *implement an $(n+1)$-way rendezvous operation abstractly (i.e., as an event value).*

For **CML**, which provides 2-way rendezvous primitives, this means that it is impossible to construct an event-valued implementation of 3-way rendezvous.

While the technical arguments are beyond the scope of this paper, the intuition behind them is fairly straightforward. An n-way rendezvous mechanism is a way for n processes to simultaneously agree. For example, in a synchronous message transmission, both the sender and acceptor agree to the transmission. Another way of viewing this is that the processes involved in a n-way rendezvous achieve some form of *common knowledge*.[4] The intuition behind the theorem is that if a language provides a primitive mechanism for n-way agreement (or n-way common knowledge), there is no way get $(n+1)$ processes to agree (attain common knowledge) in an abstract fashion; i.e., in a way that is independent of context. The problem is that while one can get n of the processes in agreement, the $(n+1)$th process is not constrained to make the same choice.

One of the immediate conclusions from these results is that there is a real difference in the expressiveness of synchronous and asynchronous message passing. A language based on asynchronous message passing, even if it supports first-class synchronous operations, cannot support abstract implementations of synchronous message passing. For the programmer, this means that situations that require both synchronous message passing and selective communication will have to be programmed on a case-by-case basis. It is interesting to note, however, that if the language does not have generalized selective communication, then one can abstractly implement synchronous communication using a simple acknowledgement-based protocol. This is because the non-blocking send operation is not a synchronous operation, and thus cannot be placed in a selective

[4] The strength of the common knowledge achieved depends on the form of the rendezvous mechanism. For example, a primitive notion of atomic rendezvous achieves common knowledge in the sense of Halpern and Moses [HM90], while a rendezvous protocol achieves the weaker *concurrent common knowledge* [PT92].

communication context. From this, we conclude that generalized selective communication is strictly more expressive than the input-only form found in many languages. Furthermore, given a language with 2-way rendezvous, but selective communication limited to input guards only, one cannot implement a communication abstraction that would allow output guards.

6 Related work

The concurrent languages **Facile** [GMP89] and **LCS** [Ber93] support communication and synchronization is way that also allows abstraction. These language make *behaviors* (a concept taken from the process calculus world) into first-class values. In many respects these behaviors are like **CML**'s event values; they can encode basic communication, choice, and actions. Behaviors are also used to implement multiple threads of control, which is their main difference from **CML**'s events. The implementations of these languages are both based on extending **ML**, and in practice, programming in these languages is likely to be quite similar to **CML** programming.

7 Conclusion

This paper has given an overview of first-class synchronous operations, along with some illustrative examples using the higher-order concurrent language **CML**. In addition to illustrating how first-class synchronous operations can be used to implement synchronization abstractions, we have also discussed their limitations, which are a result of fundamental properties of rendezvous and selective communication.

References

[AB86] Abramsky, S. and R. Bornat. Pascal-m: A language for loosely coupled distributed systems. In Y. Paker and J.-P. Verjus (eds.), *Distributed Computing Systems*, pp. 163–189. Academic Press, New York, N.Y., 1986.

[AM91] Appel, A. W. and D. B. MacQueen. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming*, vol. 528 of *Lecture Notes in Computer Science*, New York, N.Y., August 1991. Springer-Verlag, pp. 1–26.

[ANP89] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, **11**(4), October 1989, pp. 598–632.

[BCGL87] Bjornson, R., N. Carriero, D. Gelernter, and J. Leichter. Linda, the portable parallel. *Technical Report YALEU/DCS/TR-520*, Department of Computer Science, Yale University, February 1987.

[Ber93] Berthomieu, B. Programming with behaviors in an ML framework — the syntax and semantics of LCS. *Technical Report 93133*, LASS/CNRS, April 1993.

[BHLM94] Biagiono, E., R. Harper, P. Lee, and B. G. Milnes. Signatures for a network protocol stack: A systems application of Standard ML. In *Conference record of the 1994 ACM Conference on Lisp and Functional Programming*, June 1994, pp. 55–64.

[BNA91] Barth, P., R. S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In *Functional Programming Languages and Computer Architecture*, vol. 523 of *Lecture Notes in Computer Science*, New York, N.Y., August 1991. Springer-Verlag, pp. 538–568.

[Bor86] Bornat, R. A protocol for generalized occam. *Software – Practice and Experience*, **16**(9), September 1986, pp. 783–799.

[Bur88] Burns, A. *Programming in occam 2*. Addison-Wesley, Reading, Mass., 1988.

[Car86] Cardelli, L. Amber. In *Combinators and Functional Programming Languages*, vol. 242 of *Lecture Notes in Computer Science*, New York, N.Y., July 1986. Springer-Verlag, pp. 21–47.

[CG90] Carriero, N. and D. Gelernter. *How to Write Parallel Programs: A First Course*. The MIT Press, Cambridge, Mass., 1990.

[CP85] Cardelli, L. and R. Pike. Squeak: A language for communicating with mice. In *SIGGRAPH '85*, July 1985, pp. 199–204.

[DoD83] *Reference Manual for the Ada Programming Language*, January 1983.

[GB82] Gelernter, D. and A. J. Bernstein. Distributed communication via global buffer. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1982, pp. 10–18.

[Gel85] Gelernter, D. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, **7**(1), January 1985, pp. 80–112.

[GM92] Griffin, T. and R. Moten. Tactic trees in eXene. Included in the eXene distribution, 1992.

[GMP89] Giacalone, A., P. Mishra, and S. Prasad. Facile: A symemetric integration of concurrent and functional programming. In *TAPSOFT'89 (vol. 2)*, vol. 352 of *Lecture Notes in Computer Science*, New York, N.Y., March 1989. Springer-Verlag, pp. 184–209.

[GR92] Gansner, E. R. and J. H. Reppy. A foundation for user interface construction. In B. A. Myers (ed.), *Languages for Developing User Interfaces*, pp. 239–260. Jones & Bartlett, Boston, Mass., 1992.

[GR93] Gansner, E. R. and J. H. Reppy. *A Multi-threaded Higher-order User Interface Toolkit*, vol. 1 of *Software Trends*, pp. 61–80. John Wiley & Sons, 1993.

[Haa90] Haahr, D. Montage: Breaking windows into small pieces. In *USENIX Summer Conference*, June 1990, pp. 289–297.

[HM90] Halpern, J. Y. and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, **37**(3), July 1990, pp. 549–587. An earlier version appeared in the *Proceedings of the 3rd ACM Conference on Principles of Distributed Computing*.

[Kru93] Krumvieda, C. D. *Distributed ML: Abstractions for Efficient and Fault-Tolerant Programming*. Ph.D. dissertation, Department of Computer Science, Cornell University, Ithaca, NY, August 1993. Available as Technical Report TR 93-1376.

[KSI94] Kawaguchi, N., T. Sakabe, and Y. Inagaki. TERSE: Term rewriting support environment. In *Proceedings of the 1994 ACM SIGPLAN Workshop on ML and its Applications*, June 1994, pp. 91–100.

[Lin91] Lin, H. PAM: A process algebra manipulator. In *Proceedings of the Third Workshop on Computer Aided Verification*, July 1991.

[MTH90] Milner, R., M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Mass, 1990.

[Pau91] Paulson, L. C. *ML for the Working Programmer.* Cambridge University Press, New York, N.Y., 1991.

[Pik89a] Pike, R. A concurrent window system. *Computing Systems*, **2**(2), 1989, pp. 133–153.

[Pik89b] Pike, R. Newsqueak: A language for communicating with mice. *Technical Report 143*, AT&T Bell Laboratories, April 1989.

[PR94] Panangaden, P. and J. H. Reppy. On the relative expressiveness of rendezvous primitives. unpublished draft, 1994.

[PT92] Panangaden, P. and K. Taylor. Concurrent common knowledge: Defining agreement for asynchronous systems. *Distributed Computing*, 1992, pp. 73–93.

[Rep88] Reppy, J. H. Synchronous operations as first-class values. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, June 1988, pp. 250–259.

[Rep91] Reppy, J. H. CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991, pp. 293–305.

[Rep92] Reppy, J. H. *Higher-order concurrency.* Ph.D. dissertation, Department of Computer Science, Cornell University, Ithaca, NY, January 1992. Available as Technical Report TR 92-1285.

[Rep95] Reppy, J. H. *Concurrent Programming in ML.* Cambridge University Press, 1995. (forthcoming).

[RG86] Reppy, J. H. and E. R. Gansner. A foundation for programming environments. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, December 1986, pp. 218–227.

[SC91] Siegel, E. H. and E. C. Cooper. Implementing distributed Linda in Standard ML. *Technical Report CMU-CS-91-151*, cmu-cs, October 1991.

[Sli90] Slind, K. An implementation of higher order logic. Master's dissertation, University of Calgary, Department of Computer Science, December 1990.

[Tof90] Tofte, M. Type inference for polymorphic references. *Information and Computation*, **89**, 1990, pp. 1–34.

[Ull94] Ullman, J. D. *Elements of ML Programming.* Prentice Hall, Englewood Cliffs, NJ, 1994.

This article was processed using the LaTeX macro package with LLNCS style