Toward the optimization of Concurrent ML*

John Reppy
University of Chicago

October 2005

*Joint work with Yingqi Xiao

Quick CML review

Basic features:

- Explicit threading with preemptive scheduling.
- Threads communicate and synchronize via message passing using a variety of primitives (buffered channels, I-variables, and M-variables).
- Synchronization and communication are supported by the mechanism of *first-class synchronous operations* (called *events*).

Interprocess communication

```
type 'a chan

val channel : unit -> 'a chan
val recv : 'a chan -> 'a
val send : ('a chan * 'a) -> unit
```

Sending a message is a blocking operation in CML.

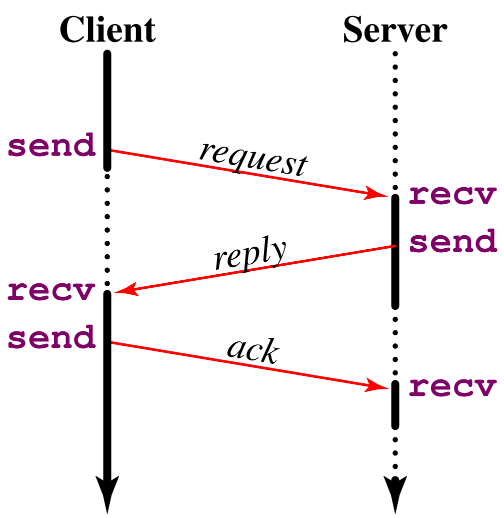
Most interactions between processes involve multiple messages.

A process may need to interact with multiple partners (*nondeterministic choice*).

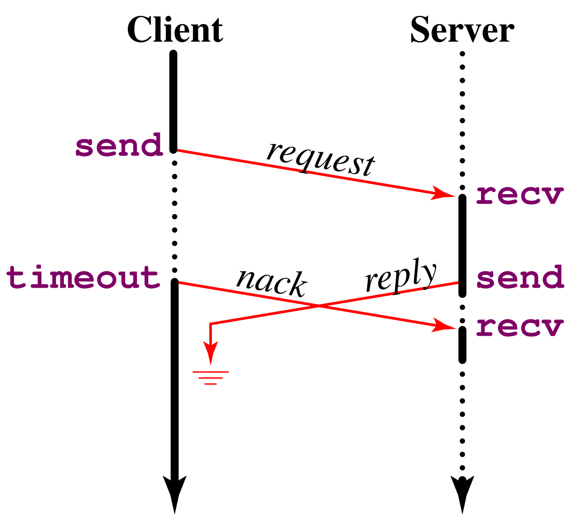
Protocols *(continued ...)*

Here are message sequence diagrams for a *client/server* protocol with acknowledgments.

Client commits



Client aborts



Events

We use *event* values to package up protocols as abstractions.

An event is an abstraction of a synchronous operation, such as receiving a message or a timeout.

```
type 'a event
```

Base-event constructors create event values for communication primitives:

```
val recvEvt : 'a chan -> 'a event
```

Events allow complicated communication protocols to be implemented as first-class abstractions.

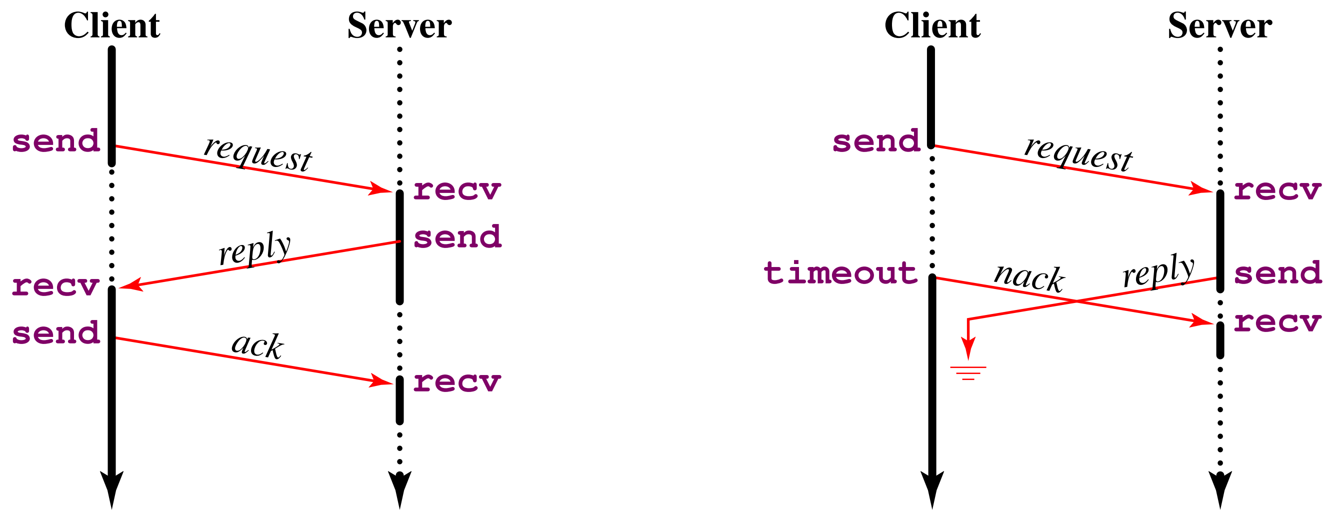
Events *(continued ...)*

CML event operations:

- Event wrappers for post-synchronization actions.
- Event generators for pre-synchronization actions and cancellation.
- Choice for managing multiple communications.
- Synchronization on an event value.

Example — client/server protocol

Recall the client/server protocol from before.



Using events, we can package it with the following abstract interface:

```
type serv
val new : () -> serv
val call : (serv * request) -> reply event
```

where `request` and `reply` are the argument and result types.

Opportunities for optimization

A couple of observations about CML in practice:

- CML communication primitives have *general* implementations (multi-party, choice, multiple messages), but a given dynamic instance of a primitive often has a *restricted* usage pattern.

- CML programs and libraries often use *abstraction* to localize a family of instances.

Channel specialization

CML communication primitives have *general* implementations (multi-party, choice, multiple messages), but a given dynamic instance of a primitive often has a *restricted* usage pattern.

For example, we can classify channels by the number of threads that might perform an operation on the channel.

number of senders	**number of** receivers	**number of** messages	**topology**
≤ 1	≤ 1	≤ 1	one-shot
≤ 1	≤ 1	> 1	point-to-point
≤ 1	> 1	> 1	one-to-many (fan-out)
> 1	≤ 1	> 1	many-to-one (fan-in)
> 1	> 1	> 1	many-to-many

Use in a choice context (or not) is another property of interest.

Channel specialization *(continued ...)*

Does exploiting this patterns gain anything?

For the current implementation of CML, we know that *one-shot* channels can be replaced by I-variables for a big improvement.

For the other patterns, the benefits are less clear in the current single-threaded implementation.

For distributed or multithreaded implementations, we expect benefit from using these specialized operations (see Demaine 1998).

Example: a simple server

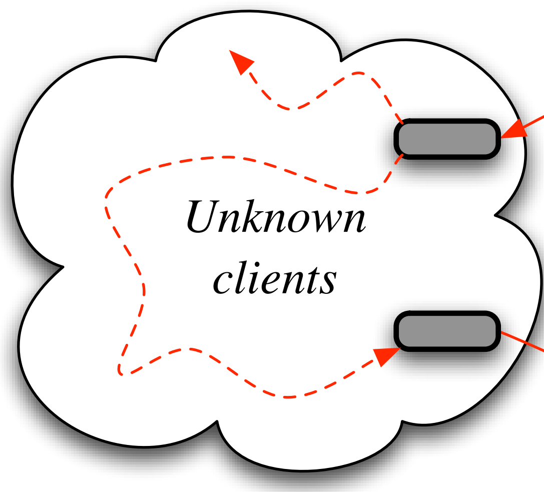
Consider a simple service that holds an integer key and that provides an operation for swapping the key.

```
signature SIMPLE_SERV =
  sig
    type serv
    val new : unit -> serv
    val call : (serv * int) -> int
  end
```

Example: a simple server *(continued ...)*

```
structure SimpleServ : SIMPLE_SERV =
  struct
    datatype serv = S of (int * int chan) chan
    fun new () = let
          val ch = channel()
          fun server v = let
                val (req, replCh) = recv ch
                in
                  send (replCh, v);
                  server req
                end
          in
            spawn (server 0);
            S ch
          end
    fun call (S ch, v) = let
          val replCh = channel()
          in
            send (ch, (v, replCh));
            recv replCh
          end
  end
```

Example: a simple server *(continued ...)*




```
fun new () = let
      val ch = channel()
      fun server v = let
            val (req, replCh) = recv ch
            in
              send(replCh, v);
              server req
            end
      in
        spawn (server 0);
       S ch
      end

fun call (S ch, v) = let
      val replCh = channel()
      in
        send (ch, (v, replCh));
        recv replCh
      end
```

Example: a simple server *(continued ...)*

```
structure SimpleServ : SIMPLE_SERV = struct
    datatype serv = S of (int * int OneShot.chan) FanIn.chan
    fun new () = let
          val ch = FanIn.channel()
          fun server v = let
                val (req, replCh) =
                       FanIn.recv ch
                in
                  OneShot.send(replCh, v);
                  server req
                end
          in
            spawn (server 0);
            S ch
          end
    fun call (S ch, v) = let
          val replCh = OneShot.channel()
          in
            FanIn.send (ch, (v, replCh));
            OneShot.recv replCh
          end
  end
```

Analysis

The hard part is knowing when it is safe to replace channels and channel operations with specialized versions.

To understand this problem, we consider a subset of CML that has **`abstype`** declarations (instead of modules), threads and channel, send and receive operations, and a monomorphic type system.

Terms in this language are annotated with *unique labels* that denote their program point.

Dynamic semantics

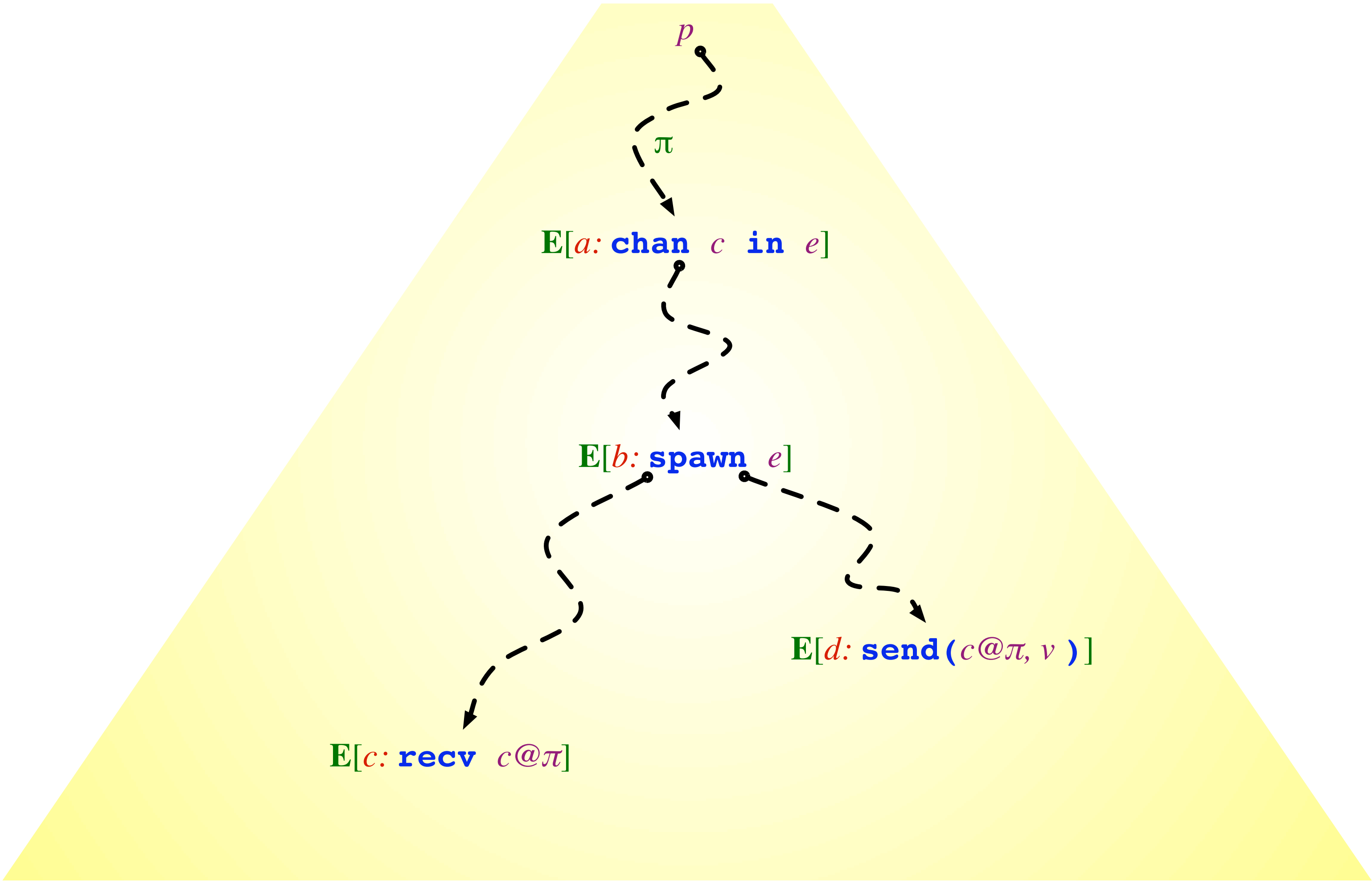
A program state is a tree (called a *trace*), where the leaves are terms that represent the current state of the threads and the path from the root to a leaf represents the history of that thread in that execution.

A small-step semantics defines how we add children to the leaves. The **`spawn`** operation adds two children to a leaf. Communication adds a single child to two leaves (the sender and the receiver).

Threads are named by the path to their **`spawn`** site in the trace. Likewise, channel *instances* are named by the path to their creation site (*e.g.* $c@\pi$).

We say that $\pi \preceq \pi'$ if π is a prefix of π'.

Semantics *(continued ...)*



Semantics *(continued ...)*

We can state our channel classification in terms of traces.

For a program p, $\mathsf{Trace}(p)$ is the set of possible finite traces.

For a trace t and channel instance k, we define

$$\begin{aligned} \mathsf{Sends}_t(k) &= \{\pi \mid t.\pi = E[\mathbf{send}(k,\, v)]\} \\ \mathsf{Recvs}_t(k) &= \{\pi \mid t.\pi = E[\mathbf{recv}\ k]\} \end{aligned}$$

We say that a channel c defined in a program p has the *single-sender property* if for any $t \in \mathsf{Trace}(p)$ and instance $c@\pi$ of c occurring in t, if $\pi_1, \pi_2 \in \mathsf{Sends}_t(c@\pi)$, then either $\pi_1 \preceq \pi_2$ or $\pi_2 \preceq \pi_1$.

The *single-receiver property* is defined similarly.

Semantics *(continued ...)*

For a channel identifier c in a program p, we can classify its topology as follows:

- The channel c is a *one-shot* channel if for any $t \in \mathtt{Trace}(p)$ and $k = c@\pi$ occurring in t, $|\mathsf{Sends}_t(k)| \leq 1$.

- The channel c is *point-to-point* if it has both the single-sender and single-receiver properties.

- The channel c is a *fan-out* channel if it has the single-sender property, but not the single-receiver.

- The channel c is a *fan-in* channel if it has the single-receiver property, but not the single-sender.

An analysis algorithm

Our analysis processes one module (abstype) at a time. It is organized into three steps:

1. A modular, *type-sensitive*, CFA based on Serrano's version of 0-CFA.
2. Construct an *extended CFG* for the module.
3. Analyze the extended CFG to determine a *safe approximation* of the communication topology.

The analysis can distinguish between multiple threads created at the same static location.

The simple server

We'll illustrate the analysis using the simple server example.

```
a1 : fun new () = (
a2 :   chan ch in
a3 :   fun server v = (
a4 :     let (w', replCh') = recv ch in
a5 :       send (replCh', v);
a6 :       server w')
       in
a7 :     spawn (a8 : server 0);
a9 :     S ch)

a10 : fun call (s, w) = (
a11 :   let S ch' = s in
a12 :   chan replCh in
a13 :     send (ch, (w, replCh));
a14 :     recv replCh)
```

Type-sensitive CFA

The CFA computes approximations of the call sites of functions and the send and receive sites of channels.

$$
\begin{aligned}
\widehat{\mathsf{SendSites}}(\mathtt{ch}) &= \{a_{13}\} \\
\widehat{\mathsf{RecvSites}}(\mathtt{ch}) &= \{a_{4}\} \\
\widehat{\mathsf{SendSites}}(\mathtt{replCh}) &= \{a_{5}\} \\
\widehat{\mathsf{RecvSites}}(\mathtt{replCh}) &= \{a_{14}\}
\end{aligned}
$$

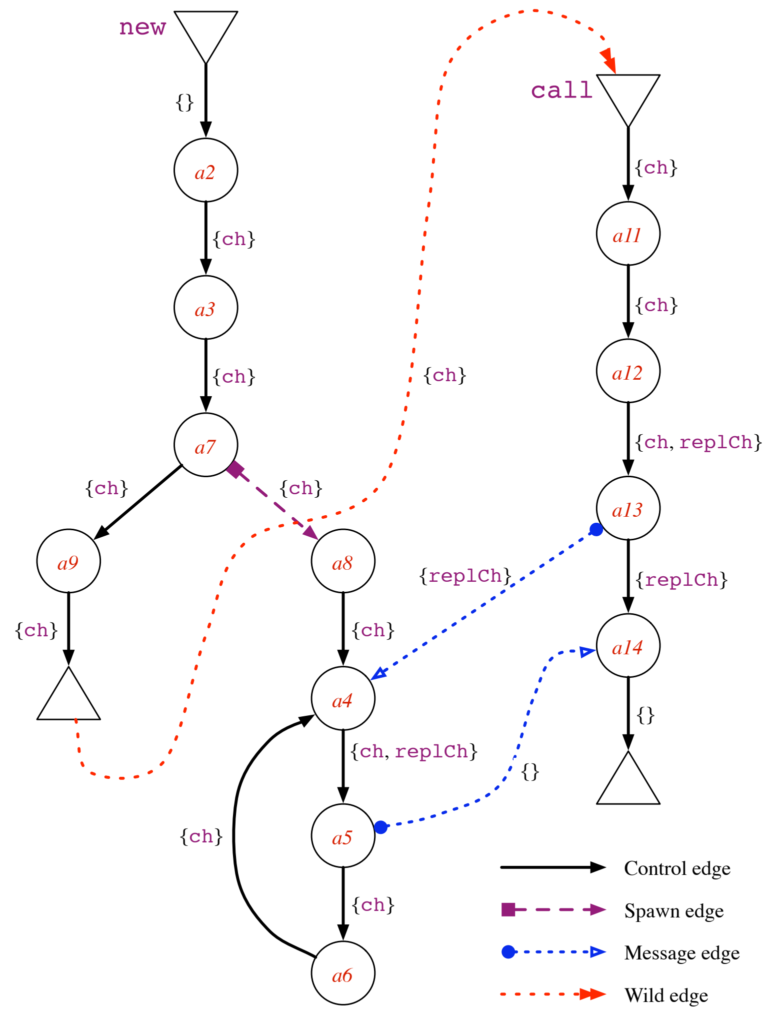
Note that even though `new` and `call` are escaping functions and `ch` escapes into the wild, the analysis is able to come up with useful information.

Extended CFA

We use the results of the CFA to construct an extended CFG.

The CFG has edges for: control-flow, spawning, messages sent from known sites to known receivers, and *wild* edges.

We label edges with the live *known* channels.



CFG analysis

We use the CFG to compute an approximation of the paths from where an instance of a channel c is created to its use sites. These paths are split into a *process ID* part and a path part. The special ID $*$ represents more than one process.

From the path approximation, we compute the sets of sender ($\widehat{S_c}$) and receiver ($\widehat{R_c}$) paths for c.

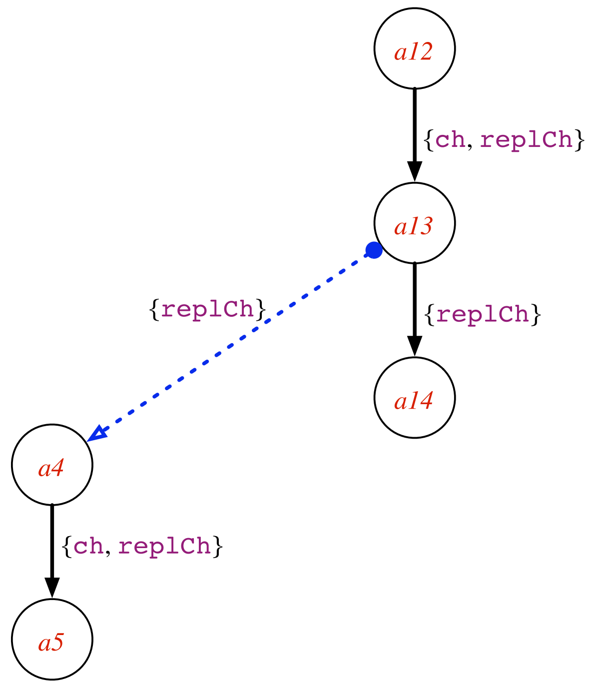
We define *approximate* single-sender/single-receiver properties in terms of $\widehat{S_c}$ and $\widehat{R_c}$.

These properties imply a safe classification of channels.

We restrict the analysis to the relevant sub-CFG.

CFG analysis *(continued ...)*

$$
\begin{aligned}
\widehat{P}_{\texttt{replCh}}(a_{12}) &= \{\epsilon{:}\epsilon\} \\
\widehat{P}_{\texttt{replCh}}(a_{13}) &= \{\epsilon{:}a_{12}\} \\
\widehat{P}_{\texttt{replCh}}(a_{14}) &= \{\epsilon{:}a_{12}a_{13}\} \\
\widehat{P}_{\texttt{replCh}}(a_{4}) &= \{a_{13}{:}\epsilon\} \\
\widehat{P}_{\texttt{replCh}}(a_{5}) &= \{a_{13}{:}a_{4}\} \\
\\
\widehat{S_{\texttt{replCh}}} &= \widehat{P}_{\texttt{replCh}}(a_5) \\
&= \{a_{13}{:}a_4\} \\
\widehat{R_{\texttt{replCh}}} &= \widehat{P}_{\texttt{replCh}}(a_{14}) \\
&= \{\epsilon{:}a_{12}a_{13}\}
\end{aligned}
$$



Thus, `replCh` is a one-shot channel.

CFG analysis *(continued ...)*

The analysis for ch is more involved, since there are loops, spawns, and wild edges involved.

The result is

$$\begin{aligned} \widehat{S_{\mathsf{ch}}} &= \{*{:}a_{11}a_{12}\} \\ \widehat{R_{\mathsf{ch}}} &= \{\pi{:}a_8,\ \pi{:}a_8a_4a_5a_6\} \end{aligned}$$

where $pi = a_2a_3\bar{a}_7$.

Thus, ch has the approximate single receiver property, but not the single-sender property, and can be implemented using a *fan-in* channel.

TODO

- Typed-based CFA as an alternative to our abstract interpretation style algorithm.
- Correctness proofs (should we use a proof assistant?)
- Other properties: no choice; single-threaded servers; ...
- Extend CFA to include event types and combinators
- Extend CFA to modules
- Implementation.