Programming Idioms for Transactional Events

Matthew Kehrt, Laura Effinger-Dean, Michael Schmitz, Dan Grossman

(University of Washington)

PLACES 2009

This Talk

- Two closely related paradigms for concurrent programming
  - Concurrent ML (CML)
  - Transactional Events (TE)
- Our experiences discovering unexpected differences

Talk Overview

- CML based on synchronous message passing
- TE adds a mechanism for *transactionally* sequencing communications to CML
  - Intuitively, such sequenced communications either all succeed or all fail

Talk Overview

- Languages must have three things:
  - Semantics
  - Implementation
  - Idioms

Talk Overview

- Languages must have three things:

	CML	TE
Semantics		
Implementation		
Idioms		

Talk Overview

- Languages must have three things:

	CML	TE
Semantics	✓	
Implementation	✓	
Idioms	✓	

Talk Overview

- Languages must have three things:

	CML	TE
Semantics	✓	✓
Implementation	✓	✓
Idioms	✓	

Talk Overview

- Languages must have three things:

	CML	TE
Semantics	✓	✓
Implementation	✓	✓
Idioms	✓	?

Contents

- Background
  - CML
  - Transactional Events
- Message Passing Idioms
  - Sequences of communications
  - Server threads

Contents

- Background
  - CML
  - Transactional Events
- Message Passing Idioms
  - Sequences of communications
  - Server threads

Concurrent ML

- Introduced by Reppy in 1991 for Standard ML
- Since been ported to functional several languages
- Lightweight threads
- Synchronous message passing

CML Channels

- CML threads use *channels* to communicate.
  - Threads send or receive on channels.
  - Sends and receives are blocking.
- Channels are first-class program values.

CML Events

- *Events* describe communications without performing them.
- An `‘a event` describes communication that yields an `‘a` when performed.
- Events are first class program values.

CML API

```
sync : ‘a event -> ‘a
sendEvt :‘a chan -> ‘a -> unit event
recvEvt : ‘a chan -> ‘a event
chooseEvt :
  ‘a event -> ‘a event -> ‘a event
wrapEvt :
  ‘a event -> (‘a -> ‘b) -> ‘b event
alwaysEvt : ‘a -> ‘a event
neverEvt : ‘a event
```

sync

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

sendEvt :
  'a chan -> 'a -> unit event

recvEvt : 'a chan -> 'a event

alwaysEvt : 'a -> 'a event

neverEvt : 'a event
```

```
chooseEvt :
  'a event -> 'a event -> 'a
event

wrapEvt :
  'a event -> ('a -> 'b)
   -> 'b event
```

- `sync` performs ("synchronizes on") an event.
  - Actually performs communication
  - May *succeed* or *fail* (block forever)

sendEvt

```
sync : ‘a event -> ‘a

sendEvt :
  ‘a chan -> ‘a -> unit event

recvEvt : ‘a chan -> ‘a event

alwaysEvt : ‘a -> ‘a event

neverEvt : ‘a event
```

```
chooseEvt :
  ‘a event -> ‘a event -> ‘a
event

wrapEvt :
  ‘a event -> (‘a -> ‘b)
   -> ‘b event
```

- `sendEvt` *chan val* creates an event that sends *val* on *chan*.

recvEvt

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

sendEvt :
  'a chan -> 'a -> unit event

recvEvt : 'a chan -> 'a event

alwaysEvt : 'a -> 'a event

neverEvt : 'a event
```

```
chooseEvt :
  'a event -> 'a event -> 'a
event

wrapEvt :
  'a event -> ('a -> 'b)
   -> 'b event
```

- `recvEvt` *chan* creates an event that receives on *chan*

Example

let c = newChan () in	
Thread 1:	Thread 2:

`sync`
`  (sendEvt c 5)`
	`let re =`
	`  recvEvt c in`
	`sync re`

chooseEvt

```
sync : ‘a event -> ‘a

sendEvt :
  ‘a chan -> ‘a -> unit event

recvEvt : ‘a chan -> ‘a event

alwaysEvt : ‘a -> ‘a event

neverEvt : ‘a event
```

```
chooseEvt :
  ‘a event -> ‘a event -> ‘a
event

wrapEvt :
  ‘a event -> (‘a -> ‘b)
   -> ‘b event
```

- **`chooseEvt`** *e1 e2* performs exactly one of *e1* or *e2*

wrapEvt

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

sendEvt :
  'a chan -> 'a -> unit event

recvEvt : 'a chan -> 'a event

alwaysEvt : 'a -> 'a event

neverEvt : 'a event
```

```
chooseEvt :
  'a event -> 'a event -> 'a
event

wrapEvt :
  'a event -> ('a -> 'b)
   -> 'b event
```

- `wrapEvt` *e f* calls *f* on the result of synchronizing on *e*.

Example

```
sync (chooseEvt
       (wrapEvt (recvEvt c1)
         (fun x ->
           (x, sync (recvEvt c2))))
       (wrapEvt (recvEvt c2)
         (fun x ->
           (sync (recvEvt c1), x))))
```

Example

```
sync (chooseEvt
       (wrapEvt (recvEvt c1)
         (fun x ->
           (x, sync (recvEvt c2))))
       (wrapEvt (recvEvt c2)
         (fun x ->
           (sync (recvEvt c1), x))))
```

Example

```
sync (chooseEvt
       (wrapEvt (recvEvt c1)
         (fun x ->
           (x, sync (recvEvt c2))))
       (wrapEvt (recvEvt c2)
         (fun x ->
           (sync (recvEvt c1), x))))
```

Example

```
sync (chooseEvt
        (wrapEvt (recvEvt c1)
          (fun x ->
            (x, sync (recvEvt c2))))
        (wrapEvt (recvEvt c2)
          (fun x ->
            (sync (recvEvt c1), x))))
```

Example

```
sync (chooseEvt
       (wrapEvt (recvEvt c1)
         (fun x ->
           (x, sync (recvEvt c2))))
       (wrapEvt (recvEvt c2)
         (fun x ->
           (sync (recvEvt c1), x))))
```

Example

```
sync (chooseEvt
       (wrapEvt (recvEvt c1)
         (fun x ->
           (x, sync (recvEvt c2))))
       (wrapEvt (recvEvt c2)
         (fun x ->
           (sync (recvEvt c1), x))))
```

Example

```
sync (chooseEvt
       (wrapEvt (recvEvt c1)
         (fun x ->
           (x, sync (recvEvt c2))))
       (wrapEvt (recvEvt c2)
         (fun x ->
           (sync (recvEvt c1), x))))
```

Example

```
sync (chooseEvt
       (wrapEvt (recvEvt c1)
         (fun x ->
           (x, sync (recvEvt c2))))
       (wrapEvt (recvEvt c2)
         (fun x ->
           (sync (recvEvt c1), x))))
```

Other Events

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

sendEvt :
   'a chan -> 'a -> unit event

recvEvt : 'a chan -> 'a event

alwaysEvt : 'a -> 'a event

neverEvt : 'a event
```

```
chooseEvt :
   'a event -> 'a event -> 'a
event

wrapEvt :
   'a event -> ('a -> 'b)
    -> 'b event
```

- `alwaysEvt` *val* succeeds immediately with *val*
- `neverEvt` never succeeds
- Both useful with `chooseEvt` and `wrapEvt`

CML Design

- CML events have at most one communication per `sync`
- Finding matching communications is fast.
- Limits expressiveness
  - Only two way synchronization

Contents

- Background
  - CML
  - Transactional Events
- Message Passing Idioms
  - Sequences of communications
  - Server threads

Transactional Events

- Based on CML
- Introduced by Donnelly and Fluet in 2006 for Haskell
- Ported to ML and substantially extended by Effinger-Dean *et al.* in 2008

`thenEvt`

```
thenEvt :
  ‘a event -> (‘a -> ‘b event) -> ‘b event
```

- TE replaces `wrapEvt` with `thenEvt`.
- Sequences events into a *transactional* event

Example

```
let c1 = newChan () in
let c2 = newChan () in
```

Thread 1:

```
let e1 =
  sendEvt c1 1 in
let e2 =
  sendEvt c2 2 in
sync
  (thenEvt e1
     (fun _ -> e2))
```

Thread 2:

```
let e =
  recvEvt
    c1 in
sync e
```

Thread 3:

```
let e =
  recvEvt
    c2 in
sync e
```

Example

```
let c1 = newChan () in
let c2 = newChan () in
```

Thread 1:	Thread 2:	Thread 3:

`let e1 =`
`  sendEvt c1 1 in`
`let e2 =`
`  sendEvt c2 2 in`
`sync`
`  (thenEvt e1`
`     (fun _ -> e2))`
	`let e =`
	`  recvEvt`
	`    c1 in`
	`sync e`
		`let e =`
		`  recvEvt`
		`    c2 in`
		`sync e`

Example

```
let c1 = newChan () in
let c2 = newChan () in
```

Thread 1:	Thread 2:	Thread 3:

`let e1 =`
`  sendEvt c1 1 in`
`let e2 =`
`  sendEvt c2 2 in`
`sync`
`  (thenEvt e1`
`     (fun _ -> e2))`
	`let e =`
	`  recvEvt`
	`    c1 in`
	`sync e`
		`let e =`
		`  recvEvt`
		`    c2 in`
		`sync e`

Example

`let c1 = newChan () in` `let c2 = newChan () in`	
Thread 1:	Thread 2:

`let e1 =`
`  sendEvt c1 1 in`
`let e2 =`
`  sendEvt c2 2 in`
`sync`
`  (thenEvt e1`
`     (fun _ -> e2))`
	`let e1 = recvEvt c1`
	`in`
	`sync e1;`
	
	`let e2 = recvEvt c2`
	`in`
	`sync e2`

Example

```
let c1 = newChan () in
let c2 = newChan () in
```

Thread 1:

```
let e1 =
  sendEvt c1 1 in
let e2 =
  sendEvt c2 2 in
sync
  (thenEvt e1
     (fun _ -> e2))
```

Thread 2:

```
let e =
  recvEvt
    c1 in
sync e
```

Thread 3:

```
let e =
  recvEvt
    c2 in
sync e
```

Transactional Events

- Synchronizing on an event is now complex.
  - Many threads
  - Many communications per **`sync`**
- Set of succeeding calls to **`sync`** is a *transaction*.

Transactional Events

- TE events can encode complex protocols.
  - *n*-way synchronization possible
  - guarded receive
- Much more complex: Exponential

Guarded Receive Example

```
sync
  (thenEvt
    (recvEvt c1) (fun x ->
    (recvEvt c2) (fun y ->
    if y > x
    then alwaysEvt (x,y)
    else neverEvt))))
```

The Question

- `thenEvt` can express communications that `wrapEvt` cannot.
- Can `wrapEvt` express things `thenEvt` cannot?
- More concretely, can we use existing CML idioms in TE?

Contents

- Background
  - CML
  - Transactional Events
- Message Passing Idioms
  - Sequences of communications
  - Server threads

CML `wrapEvt` Example

```
sync (chooseEvt
       (wrapEvt (recvEvt c1)
         (fun x ->
           (x, sync (recvEvt c2))))
       (wrapEvt (recvEvt c2)
         (fun x ->
           (sync (recvEvt c1), x))))
```

TE `thenEvt` Example

```
sync (chooseEvt

   (thenEvt (recvEvt c1) (fun x ->
    thenEvt (recvEvt c2) (fun y ->
    alwaysEvt (x,y))))

   (thenEvt (recvEvt c2) (fun y ->
    thenEvt (recvEvt c1) (fun x ->
    alwaysEvt (x,y)))))
```

TE `thenEvt` Example

```
sync (chooseEvt

   (thenEvt (recvEvt c1) (fun x ->
    thenEvt (recvEvt c2) (fun y ->
    alwaysEvt (x,y))))

   (thenEvt (recvEvt c2) (fun y ->
    thenEvt (recvEvt c1) (fun x ->
    alwaysEvt (x,y)))))
```

TE `thenEvt` Example

```
sync (chooseEvt

  (thenEvt (recvEvt c1) (fun x ->
   thenEvt (recvEvt c2) (fun y ->
   alwaysEvt (x,y))))

  (thenEvt (recvEvt c2) (fun y ->
   thenEvt (recvEvt c1) (fun x ->
   alwaysEvt (x,y)))))
```

TE `thenEvt` Example

```
sync (chooseEvt

  (thenEvt (recvEvt c1) (fun x ->
   thenEvt (recvEvt c2) (fun y ->
   alwaysEvt (x,y))))

  (thenEvt (recvEvt c2) (fun y ->
   thenEvt (recvEvt c1) (fun x ->
   alwaysEvt (x,y)))))
```

TE `thenEvt` Example

```
sync (chooseEvt

  (thenEvt (recvEvt c1) (fun x ->
   thenEvt (recvEvt c2) (fun y ->
   alwaysEvt (x,y))))

  (thenEvt (recvEvt c2) (fun y ->
   thenEvt (recvEvt c1) (fun x ->
   alwaysEvt (x,y)))))
```

TE `thenEvt` Example

```
sync (chooseEvt

  (thenEvt (recvEvt c1) (fun x ->
   thenEvt (recvEvt c2) (fun y ->
   alwaysEvt (x,y))))

  (thenEvt (recvEvt c2) (fun y ->
   thenEvt (recvEvt c1) (fun x ->
   alwaysEvt (x,y))))))
```

TE `thenEvt` Example

```
sync (chooseEvt

  (thenEvt (recvEvt c1) (fun x ->
   thenEvt (recvEvt c2) (fun y ->
   alwaysEvt (x,y))))

  (thenEvt (recvEvt c2) (fun y ->
   thenEvt (recvEvt c1) (fun x ->
   alwaysEvt (x,y)))))
```

CML and TE

- TE code does not reflect the CML code.
- TE will not work with

```
sync (sendEvt c1 1); sync (sendEvt c2 2)
```

- TE requires both sends in same call to `sync`.

Delaying Events

- Problem is that synchronizing on a `thenEvt` perform the second event
- Can delay events until after current call to `sync` using thunks

`wrapEvt` idiom in TE

```
(sync (chooseEvt
  (thenEvt
    (recvEvt c1)
    (fun x -> alwaysEvt (fun () ->
      (x, sync (recvEvt c2)))))
  (thenEvt
    (recvEvt c2)
    (fun x -> alwaysEvt (fun () ->
      (sync (recvEvt c1), x))))))

  ()
```

`wrapEvt` idiom in TE

```
(sync (chooseEvt
  (thenEvt
    (recvEvt c1)
    (fun x -> alwaysEvt (fun () ->
      (x, sync (recvEvt c2)))))
  (thenEvt
    (recvEvt c2)
    (fun x -> alwaysEvt (fun () ->
      (sync (recvEvt c1), x))))))

  ()
```

wrapEvt idiom in TE

```
(sync (chooseEvt
  (thenEvt
    (recvEvt c1)
    (fun x -> alwaysEvt (fun () ->
      (x, sync (recvEvt c2)))))
  (thenEvt
    (recvEvt c2)
    (fun x -> alwaysEvt (fun () ->
      (sync (recvEvt c1), x))))))

  ()
```

`wrapEvt` idiom in TE

```
(sync (chooseEvt
  (thenEvt
    (recvEvt c1)
    (fun x -> alwaysEvt (fun () ->
      (x, sync (recvEvt c2)))))
  (thenEvt
    (recvEvt c2)
    (fun x -> alwaysEvt (fun () ->
      (sync (recvEvt c1), x))))))

  ()
```

`wrapEvt` idiom in TE

```
(sync (chooseEvt
  (thenEvt
    (recvEvt c1)
    (fun x -> alwaysEvt (fun () ->
      (x, sync (recvEvt c2)))))
  (thenEvt
    (recvEvt c2)
    (fun x -> alwaysEvt (fun () ->
      (sync (recvEvt c1), x))))))

  ()
```

`wrapEvt` idiom in TE

```
(sync (chooseEvt
  (thenEvt
    (recvEvt c1)
    (fun x -> alwaysEvt (fun () ->
      (x, sync (recvEvt c2)))))
  (thenEvt
    (recvEvt c2)
    (fun x -> alwaysEvt (fun () ->
      (sync (recvEvt c1), x))))))

  ()
```

`wrapEvt` idiom in TE

- Too complex
- We can abstract the thunks.

```
let thunkWrap ev f =
  thenEvt ev
    (fun x -> alwaysEvt (fun () -> f x))

let syncThunked ev = (sync ev) ()
```

`wrapEvt` idiom in TE

```
let thunke = (chooseEvt
  (thunkWrap (recvEvt c1)
    (fun x -> (x, sync (recvEvt c2))))
  (thunkWrap (recvEvt c2)
    (fun x -> (sync (recvEvt c1), x))))
in
syncThunked thunke
```

- We have regained CML behavior, but programmers must decide where to use **`sync`** and **`syncThunked`**.

Contents

- Background
  - CML
  - Transactional Events
- Message Passing Idioms
  - Sequences of communications
  - Server threads

Servers

- Server threads a common CML idiom
- Repeatedly communicate to clients
- TE allows *clients* to have complicated protocols.
  - Want to write servers to handle any client

Server Example

```
let c = newChan () in
```

Server:	Client:

`let rec loop y =`
`  sync`
`    (sendEvt c y);`
`    loop (y + 1)`
`in`
`loop 0`
	`sync`
	`  (recvEvt c)`

Server Example

```
let c = newChan () in
```

Server:	Client:

<pre>let rec loop y =
  sync
    (sendEvt c y);
    loop (y + 1)
in
loop 0</pre>
	<pre>sync
	  (thenEvt
	    (recvEvt c)
	    (fun x ->
	        recvEvt c))</pre>

New Server

- Client can require server to make an arbitrary number of sends in one synchronization.
- Need a new TE server
  - Must be able to compute and send multiple values per transaction

New Server Design

- Two nested loops
  - Inner loop uses thenEvt and chooseEvt to send an arbitrary number of values in one transaction
  - Outer loop calls inner loop with next value
- In combination, sends stream of values with arbitrary transaction boundaries.

New Server Code

```
let rec evtLoop x =
  thenEvt (sendEvt c x)
   (fun _ -> chooseEvt
     (alwaysEvt (x + 1))
     (evtLoop (x + 1))) in

let rec serverLoop x =
  serverLoop
    (sync (evtLoop x)) in

serverLoop 0
```

Generalizing the New Server

- That was complicated.
- Paper generalizes `evtLoop` and `serverLoop` to arbitrary servers.

Lessons

- CML idioms cannot always be straightforwardly transferred to TE.
- `thenEvt` creates complications in unexpected places.

More in Paper

- Paper contains CML examples implemented with `thenEvt.`
  - `guardEvt` based timouts
  - `wrapAbort` mutex server

Conclusions

- Languages must have three things:
  - Semantics
  - Implementation
  - Set of idioms

Conclusions

- TE had implementation, semantics.
- Idioms were unexpectedly difficult.
- Thought and experience were required.

Questions?