Programming Idioms for Transactional Events

Matthew Kehrt Laura Effinger-Dean Michael Schmitz Dan Grossman

University of Washington

{mkehrt,effinger,schmmd,djg}@cs.washington.edu

Abstract

Transactional events (TE) are an extension of Concurrent ML (CML), a programming model for synchronous message-passing. Prior work has focused on TE's formal semantics and its implementation. This paper considers programming idioms, particularly those that vary unexpectedly from the corresponding CML idioms. First, we solve a subtle problem with client-server protocols in TE. Second, we argue that CML's `wrap` and `guard` primitives do not translate well to TE, and we suggest useful workarounds. Finally, we discuss how to rewrite CML protocols that use abort actions.

1 Introduction

Transactional events (TE) provide powerful message-passing facilities for concurrent programming [1]. TE extends Concurrent ML (CML) [3] with a sequencing primitive `thenEvt`, which allows an arbitrary number of sends or receives per event. `thenEvt` is powerful enough that programmers can implement patterns such as *n*-way rendezvous (synchronizing multiple threads at once) or guarded receive (successfully synchronizing only if a received value satisfies a predicate).

In previous work [1, 2], we and other researchers explored the semantics and implementation of transactional events. However, no existing work discusses practical programming idioms for TE. We have found in the course of our research that some common CML idioms are surprisingly difficult to reproduce in TE. This paper presents our solutions to these problematic idioms, which include client-server protocols and programs using CML's `wrap`, `guard`, and `wrapAbort`.

TE is implemented for Haskell and Caml. In our examples, we use Caml, which is call-by-value.

2 Background

TE [1, 2] and CML [3] are closely related paradigms for concurrent programming that have been implemented for several languages. This section briefly reviews TE and explains how it differs from CML.

In CML, threads send values on channels. A channel of type `'a chan` carries values of type `'a`. Communication is *synchronous*: a send blocks until matched with a receive in another thread.

An *event* is a description of a communication to be performed. The functions `sendEvt` and `recvEvt` produce events that describe sends and receives, respectively. *Synchronizing* on an event with the function `sync` performs the event; `sync` has type `'a event -> 'a`. Events may be composed of other events. For example, the function `chooseEvt` constructs an event that, when synchronized on, performs exactly one of two events. The types of these and other functions appear in Figure 1.

TE extends CML with a new function `thenEvt`, which allows the sequencing of events. Synchronizing on `thenEvt ev f` does the following: (1) synchronize on `ev` to produce a result `v`; (2) apply `f` to `v` to produce a new event `ev2`; (3) synchronize on `ev2` to produce a final result. Critically, these three steps are all-or-nothing: if the second event cannot successfully synchronize, then the first event does not (appear to) happen. Therefore, we say that events built using `thenEvt` are *transactional*. A single event may communicate with multiple threads, so the success of one synchronization may entail the success of an arbitrary number of synchronizations in other threads.

Two more CML/TE functions are useful in combination with `chooseEvt` and `thenEvt`. `alwaysEvt` is the event that always succeeds: `sync (alwaysEvt x)` is equivalent to `x`. `neverEvt` is the event that never succeeds: `sync neverEvt` blocks forever.

The following example tries to do either a single send, or a receive followed by a send:

```
sync (chooseEvt (sendEvt c1 5)
                (thenEvt (recvEvt c2) (fun x -> sendEvt c3 x)))
```

```
type 'a chan
type 'a event
val newChan : unit -> 'a chan
val sync : 'a event -> 'a
val sendEvt : 'a chan -> 'a -> unit event
val recvEvt : 'a chan -> 'a event
val chooseEvt : 'a event -> 'a event
                -> 'a event
val thenEvt : 'a event -> ('a -> 'b event)
              -> 'b event
val alwaysEvt : 'a -> 'a event
val neverEvt : 'a event
val guard : (unit -> 'a event)
            -> 'a event
val wrap : 'a event -> ('a -> 'b)
           -> 'b event
val wrapAbort : 'a event
                -> (unit -> unit)
                -> 'a event
```

Figure 1: Types for several key CML/TE functions.

3 Server loops

One common concurrent programming idiom is a server thread that repeatedly handles requests from multiple clients. In CML, a server is often implemented as an infinite loop that calls `sync` at every iteration. In this section, we discuss why TE requires more sophisticated servers and how to implement them.

Consider the following function, which spawns a new thread to act as a server. The server sends increasing integers on a channel so that clients get a unique integer every time they receive on the channel.

```
let simpleIncrementServer () =
  let c = newChan () in
  let rec f y = sync (sendEvt c y); f (y + 1) in
  Thread.create f 0; c
```

A simple client receives on the server's channel.

```
let simpleIncrementClient c = sync (recvEvt c)
```

Both client and server are perfectly valid as both TE and CML. However, in TE `thenEvt` can be used to write other clients that interact with this server in unexpected ways. The following code receives two integers and adds them together in a single event:

```
let complexIncrementClient c =
  sync (thenEvt (recvEvt c) (fun x ->
        thenEvt (recvEvt c) (fun y ->
        alwaysEvt (x + y))))
```

If this client and the server were to synchronize on their events, neither would succeed. The client could receive one integer from the server. The server event would block waiting for the client event to complete, as they would be participating in the same transaction. Meanwhile, the client would block waiting for another integer. Both sides would be stuck, so this particular client cannot successfully synchronize with the simple increment server.

A similar problem arises when *two* client threads each receive an integer from the server and then communicate with each other in the same event. A transaction consisting only of these two events and the server event would not succeed. The server would need to send to both threads, but the server's event does a single send.

To solve this problem, we want a server that can perform multiple sends in a single synchronization. In the TE code below, the server synchronizes on a single event that can perform an arbitrary number of sends. After each send, the event chooses between either returning the sent value or recursively calling the server function.

```
let betterIncrementServer () =
  let c = newChan () in
  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
  Thread.create serverLoop 0; c
```

However, this problem can occur with many different server and client combinations. A better solution would be to create a generic combinator for an event that is repeated one or more times.

For this purpose, we define a function, `serverLoop`, suitable for creating servers. It takes two arguments and loops forever. The first argument is a pair, `(ev, b)`, of an event and any value. The second argument is a function, `f`. After synchronizing on `ev` to produce a value `a`, `serverLoop` calls `f` on the pair `(a,b)` to produce a new `(ev,b)` pair, with which it recurs. `b` acts as a loop-carried state for `serverLoop`. Programmers can use `serverLoop` much like they use `fold` to process lists.

Internally, `serverLoop` uses a second function, `evtLoop`, that creates an event that synchronizes on `ev` and then nondeterministically chooses between returning or recurring with the result of calling `f` to produce a new event and loop-carried state. In other words, `evtLoop` does exactly what `serverLoop` does but, crucially, in a single transaction. Overall, `serverLoop` sequentially synchronizes on the events computed by `f`, but transactions may end (starting the next transaction) at any point in the sequence. Thus, clients can communicate with the server any number of times in one synchronization.

```
(* evtLoop : ('a event * 'b) -> (('a * 'b) -> ('a event * 'b)) -> ('a * 'b) event *)
let rec evtLoop (ev, b) f =
  thenEvt ev (fun a -> chooseEvt (alwaysEvt (a, b)) (evtLoop (f (a, b)) f))

(* serverLoop : ('a event * 'b) -> (('a * 'b) -> ('a event * 'b)) -> 'c *)
let rec serverLoop (ev, b) f = serverLoop (f (sync (evtLoop (ev, b) f))) f
```

Using `serverLoop` for our increment server is straightforward. We spawn a new thread to run `serverLoop` called with (1) an initial event paired with the initial loop-carried counter and (2) a function to construct the next event and counter by incrementing the counter and creating the next send event.

```
let incrementServer () =
  let c = newChan () in
  Thread.create (serverLoop ((sendEvt c 0), 0))
                            (fun (_, x) -> (sendEvt c (x+1), x+1)); c
```

4 `wrap` and `guard`

`wrap` and `guard` (see Figure 1) add post- and pre-processing, respectively, to CML events. `sync (wrap ev f)` synchronizes on `ev`, then applies `f` to the result. `sync (guard g)` synchronizes on the result of `g ()`. In this section, we discuss how to adapt programs that use these functions to TE.

The following code uses `wrap` to perform two receives in either order:

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

Suppose we were to rewrite this code using `thenEvt`:

```
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)))))
```

These two versions actually behave differently: the CML version performs the receives in separate synchronizations, while the TE version executes both in the *same* synchronization. Therefore the above

TE code could not communicate successfully with code that performed two synchronizations, such as `sync (sendEvt c1 4); sync (sendEvt c2 5)`.

We can mimic `wrap`'s behavior in TE by thunking the second receive and executing the thunk after the first `sync` completes:

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

With the use of two helper functions, the TE code approaches the elegance of the original CML code:

```
let thunkWrap ev f = thenEvt ev (fun x -> alwaysEvt (fun () -> f x))
let syncThunked ev = (sync ev) ()
let _ = syncThunked (chooseEvt
                       (thunkWrap (recvEvt c1) (fun x -> (x, sync (recvEvt c2))))
                       (thunkWrap (recvEvt c2) (fun x -> (sync (recvEvt c1), x))))
```

We have sacrificed some composability: `thunkWrap` returns a `(unit -> 'b) event` instead of a `'b event`, so it is more difficult than `wrap` to combine with other events. The semantics of `wrap` (processing an event's result after synchronization completes) and `thenEvt` (combining two synchronizations into one) seem to be incompatible, but we believe that thunked wrappers are a good compromise. Wrapping an already wrapped event does not require a second level of thunk, as this helper function demonstrates:

```
(* rewrap : (unit -> 'a) event -> ('a -> 'b) -> (unit -> 'b) event *)
let rewrap ev g = thenEvt ev (fun f -> let x = f () in alwaysEvt (fun () -> g x))
```

CML's `guard` is useful for encapsulating actions that need to happen prior to synchronization. For example, the following code adds a timeout to an event by spawning a thread to signal when to give up:

```
(* timeoutEvt : 'a event -> float -> 'a event *)
let timeoutEvt ev time = guard (fun () ->
  let timeoutChan = newChan () in
  Thread.create (fun () -> Thread.delay time; sync (sendEvt timeoutChan ())) ();
  chooseEvt ev (wrap (recvEvt timeoutChan) (fun () -> raise TimedOutExn)))
```

As with `wrap`, it is difficult to add `guard` to TE's interface because the guard function needs to execute outside of the synchronization. However, we can code up `timeoutEvt` in TE without `guard`:

```
let timeoutEvt ev time = chooseEvt ev
  (thenEvt (alwaysEvt ()) (fun () -> Thread.delay time; raise TimedOutExn))
```

Moreover, this function is more readable than the CML code. In the next section, we will see another example for which the TE implementation is more natural than the original CML program.

5 `wrapAbort` and abort actions

The `wrapAbort` function (see Figure 1) lets CML programs specify an action to take if an event is part of a `chooseEvt` that is synchronized on and another choice is taken. For example, `sync (chooseEvt (wrapAbort ev1 f) ev2)` will block until either `ev1` or `ev2` succeeds, and in the latter case it will execute `f ()`. `wrapAbort` is useful for client-server protocols that have more than one communication. The CML book [3] uses `wrapAbort` to implement mutual-exclusion locks with this interface[1]:

```
type lockServer
val acquireLockEvt : lockServer -> int -> unit event
val releaseLockEvt : lockServer -> int -> unit event
val mkLockServer : unit -> lockServer
```

[1] It actually uses the equally expressive `withNack`; we discuss `wrapAbort` because we find it more intuitive.

Clients of this interface acquire or release locks (represented by `int`s) by synchronizing on events created with `acquireLockEvt` and `releaseLockEvt`. Synchronizing on `acquireLockEvt s i` blocks until the acquire succeeds; clients may abort acquires, perhaps with the `timeoutEvt` function from Section 4:

```
sync (timeoutEvt (acquireLockEvt server 1) 5.0)
```

Implementing `acquireLockEvt` with a single server thread in CML requires two communications: a request from the client with the lock ID, and a confirmation from the server when the lock is available. Abort actions are essential for implementing `acquireLockEvt`, as the first communication may affect the server's internal state by adding the request to a queue. If the client aborts after the first communication but before the second, it must tell the server to remove the request from the queue. Otherwise, the server would hang when trying to confirm the lock acquire. The form of `acquireLockEvt` is essentially:

```
let acquireLockEvt s id =
  guard (fun () -> (* send acquire request *);
                   wrapAbort (* receive acquire confirmation *)
                             (fun () -> (* do cancellation *)))
```

The abort action effectively makes the two communications *transactional*: if the second communication aborts, the effects of the first communication are canceled. In TE, we can implement a lock server without an abort action. Our solution uses `thenEvt` and `neverEvt` and is similar to the guarded-receive pattern [1]. If the server receives a request for an unavailable lock (the server maintains a list of held locks), it returns `neverEvt` inside `thenEvt`. `neverEvt` never succeeds, so the program behaves as if the request did not yet occur, exploiting the transactional semantics of `thenEvt`. A full implementation is below[2]; we expect other CML protocols that use abort actions will adapt similarly to TE.

```
type request = Acquire of int | Release of int
type lockServer = request chan
let acquireLockEvt reqCh id = sendEvt reqCh (Acquire id)
let releaseLockEvt reqCh id = sendEvt reqCh (Release id)
let mkLockServer () =
  let reqCh = newChan () in
  let serverEvt heldLocks =
    thenEvt (recvEvt reqCh) (function
      | Acquire id -> if List.exists (fun id2 -> id = id2) heldLocks
                      then neverEvt
                      else alwaysEvt (id::heldLocks)
      | Release id -> alwaysEvt (List.filter (fun id2 -> id <> id2) heldLocks))
  in Thread.create (serverLoop (serverEvt [], ()))
       (fun (heldLocks, ()) -> (serverEvt heldLocks, ()));
     reqCh
```

6 Conclusion

Every programming model needs three things: a semantics, an implementation, and useful idioms. Reppy's dissertation on CML [3] presents all three to demonstrate that CML is a useful programming model. Prior TE research [1, 2] has concentrated on semantics and implementation. We have presented several important TE programming idioms, the subtleties of which surprised us during our research. First, writing client-server protocols in TE requires careful consideration of how `sync` interacts with `thenEvt`; our `serverLoop` function is a general solution to this problem. Second, `wrap` and `guard` are difficult to integrate with TE, and we have suggested alternatives that preserve most of the original semantics. Finally, we have discussed how protocols with abort actions may be rewritten with `thenEvt` and `neverEvt`.

[2] As an orthogonal issue, we use `serverLoop` from Section 3 to support multiple lock operations in one synchronization.

References

[1] Kevin Donnelly and Matthew Fluet. Transactional events. In *11th ACM International Conference on Functional Programming*, 2006.

[2] Laura Effinger-Dean, Matthew Kehrt, and Dan Grossman. Transactional events for ML. In *13th ACM International Conference on Functional Programming*, 2008.

[3] John Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.