Isolates: Serializability Enforcement for Concurrent ML

Lukasz Ziarek
Armand Navabi
Suresh Jagannathan

CSD TR #10-007
June 2010

Isolates: Serializability Enforcement for Concurrent ML

Lukasz Ziarek, Armand Navabi, and Suresh Jagannathan

Purdue University
{lziarek, anavabi, suresh}@cs.purdue.edu

Abstract. There has been much recent interest in exploring higher-level concurrency control abstractions such as software transactional memory (STM) to alleviate the complexity of reasoning about interactions among concurrent threads of control. *Isolation* and *atomicity* are the two critical properties provided by an STM that guarantee serializability of concurrent actions. Isolation ensures that transactions execute without interference from effects performed by other transactions, and atomicity guarantees that intermediate effects performed by a transaction are not seen by other concurrently executing transactions.

While these properties have been primarily designed with shared memory in mind, there has been recent work (5; 6) that explores how atomicity could be leveraged to increase the expressivity of message-passing abstractions such as the first-class synchronous events found in Concurrent ML (CML) (17). Notably, these proposals do *not* enforce isolation of concurrently executing events, and thus cannot be used to enforce transactional execution of CML programs. In this paper, we consider the introduction of a new event combinator that addresses this significant limitation. An *isolate* is a combinator that allows a complex event to execute in isolation with other concurrently executing events (including other isolates). By doing so, it enables the integration of a true transactional semantics into a CML-style concurrency model, enabling reasoning about CML programs in terms of serializable event orderings. Incorporating isolation into CML poses a number of challenging problems, however, whose solutions form the focus of this paper.

1 Introduction

Programming with concurrency is challenging because reasoning about nondeterministic interactions among concurrently executing computations is difficult. Concurrency control abstractions such as software transactions (15) have attracted significant interest because they simplify reasoning about these interactions. Transactions provide two key guarantees on the computations they encapsulate: (1) *isolation* ensures that a computation can execute without interference from effects performed by other threads; (2) *atomicity* ensures that intermediate effects performed within a transaction do not become visible until the transaction completes.

Recently, *transactional events* (5; 6) have been proposed as a way to leverage the power of atomicity to increase the expressivity of message-passing abstractions such as first-synchronous events found in languages like Concurrent ML (CML) (17). While CML allows the construction of complex events from base events, synchronization can only take place on a single event. This limitation makes it difficult to express certain communication protocols and impossible to express others, such as three-way rendezvous. Transactional events address this limitation by allowing multiple communication actions to be encapsulated within a new event combinator (`thenEvt`) that makes the effects of these actions visible to other threads provided all of them can succeed. For example, the expression, `thenEvt(sendEvt (c1, v), fn() => recvEvt c2)`, when synchronized will only complete if the operation executes in a state in which both the `sendEvt` and `recvEvt` can succeed in that order as a single atomic action. The guarantee of atomicity provided by `thenEvt` distinguishes it from the functionality provided by other CML event combinators, and leads to a substantial increase in expressive power and programmability.

Unfortunately, unlike mainstream software transactions, two threads concurrently executing `thenEvt`s can have their communication effects witnessed by the other; when this happens, a larger atomic unit is formed. Thus, while transactional events provide an all-or-nothing property on the events they encapsulate, they provide no mechanism to *restrict* visibility of effects performed within the dynamic context of a transactional event. Repairing this limitation forms the focus of this paper. The ability to isolate effects of different concurrently executing transactional events simplifies program reasoning, and prevents unwanted interactions. Without isolation, any effect witnessed by one transactional event performed by another effectively pairs the two together: even though failure of one results in the failure of the other, there is no facility available to prevent these intermediate actions from being observed in the first place.

To support isolation, we introduce *isolates*, a new abstraction that enforces isolation of concurrently evaluating events. The ability to execute concurrent events in isolation along with the all-or-nothing atomicity property of transactional events not only allows the construction of truly transactional (i.e., serializable) CML programs, but leads to additional expressivity and efficiency that would otherwise not be possible.

The remainder of the paper is organized as follows. The next section briefly describes CML and transactional events. Section 3 provides a motivating example. Section 4 presents a naive semantics for isolates that that imposes a simple *a priori* ordering on isolate evaluation that ensures serializability but at the price of constraining concurrency. In Section 5 we relax this restriction; our new formulation uses *capabilities* to track potential interfering communication actions, limiting concurrent evaluation only when necessary. Section 6 states soundness results that relate the two different formulations. Section 7 discusses implementation issues, and Section 8 presents related work and conclusions.

2 CML and Transactional Events

Concurrent ML (17) (CML) is a concurrent extension of Standard ML that utilizes synchronous message passing for communication among concurrently executing threads. Threads perform `send` and `recv` operations on typed channels; these operations block until a matching action on the same channel is performed by another thread.

CML also provides first-class synchronous *events* that abstract synchronous message-passing operations. An event value of type `'a event` when synchronized yields a value of type `'a`. Thus, an event value represents a potential computation, with latent effect until a thread synchronizes upon it by calling `sync`. The following equivalences thus hold: `send(c, v)` $\equiv$ `sync(sendEvt(c,v))` and `recv(c)` $\equiv$ `sync(recvEvt(c)`. Besides `sendEvt` and `recvEvt`, there are two other base events that we refer to in the paper: `alwaysEvt` given a value returns an event that when synchronized upon returns that value; `neverEvt` yields an event that can never be successfully synchronized upon.

Much of CML's expressive power derives from event combinators that construct complex event values from other events. We list some of these combinators in Fig. 1. The `chooseEvt` event combinator takes a list of events and constructs an event value that represents the non-deterministic choice of the events in the list; for example, `choosEvt[recvEvt(a),sendEvt(b,v)]` when synchronized will either receive a unit value from channel `a`, or send value `v` on channel `b`. The expression `wrap (ev, f)` creates an event that when synchronized applies the result of synchronizing on event `ev` to function `f`. Conversely, `guard(f)` creates an event which when synchronized evaluates `f()` to yield event `ev` and then acts as `ev`.

```
sendEvt : a chan * a -> unit event
recvEvt : a chan -> a event
neverEvt : a event
alwaysEvt : a -> a event
sync : a event -> a
choose : a event list -> a event
wrap : a event * (a -> b) -> b event
guard : (unit -> a event) -> a event
```

Fig. 1. CML event operators.

While CML's event combinators provide a great deal of expressivity, there are nonetheless useful abstractions that are difficult or impossible to define. One such example is a safe guarded receive; given a channel `c` and a guard `g`, the operation accepts `v` from `c` *only if* `g(v)` yields true. Expressing this functionality in CML is difficult because the act of transmitting a value from the sender to receiver requires a synchronization; once the synchronization is complete, there is no facility to revoke the communication in case the guard yields false.

To address these limitations, a *transactional event*) combinator (written as `thenEvt(evt,f)`), that sequences multiple communication actions into a single atomic event is necessary (5; 6). A transactional event takes an event `evt` and a function `f` for producing a new event from the result of the first event. A synchronization action on a `thenEvt` first (provisionally) synchronizes on `evt`, and calls `f` with the resulting value. The event yielded by the application is then synchronized on as well. If both synchronizations succeed (i.e., neither event yields `neverEvt`) the transactional event succeeds, yielding the value of the last event; if either fails, the entire event fails to synchronize, effectively erasing any of the provisional actions. Using transactional events, we can now easily express a guarded receive:

```
thenEvt(recvEvt(ch),
        fn(v) => if g(v) then alwaysEvt(v)
                         else neverEvt)
```

Because the `neverEvt` returned when `g(v)` yields false can never be synchronized upon, we ensure that the act of receiving a value from channel `ch` occurs only if `g(v)` is true. Thus, transactional events abstractly define atomic sets of communication actions through the use of the `thenEvt` combinator. Such sets are dynamically linked to create larger atomic units.

3 The Need for Isolation

Consider a server abstraction that mediates access to a pool of identical resources. The server provides three operations: (1) *query* that returns information about the resources it holds; (2) *reserve* that requests some number of these resources to a client; and (3) *release* that returns a previously reserved set of resources back to the pool. A simple implementation of the server written in CML in which resources are abstracted as integers is given in Fig. 2.

The server is implemented as a thread that loops, repeatedly waiting for input on a dedicated input channel. The server loop is implemented as a transactional event that encapsulates the act of reading the next operation from a client, and yielding the result. If a client asks to reserve a number of resources fewer than the number of available, the server indicates the success of the operation by yielding an `alwaysEvt`; if the client request is not satisfiable, a `neverEvt`, which can never be successfully synchronized against, is returned, thus preventing a client communicating with the server from within its own transactional event from committing any other actions performed by that event. The `sync` operation is successful only if the requested operation is successful, in which case the new server state is available for the next iteration.

With this interface, we might consider writing a client (see Fig. 3 that queries a server to check the number of resources it has, and based on the result, makes a request to reserve some number of them, using transactional events to ensure that the query and reservation execute atomically.

The `query` function communicates to the server to query the server's state; once known, the client uses auxiliary function `f` to reserve some number of

```
datatype ops = Query of int chan | Req of int | Rel of int

fun server(n) =
  let val reqCh = channel()
      fun serverLoop(n) =
        let val evt =
             thenEvt(recvEvt(reqCh),
                     fn (req) =>
                          case req of
                            Query ch => wrapEvt(sendEvt(ch,n),
                                                fn () => alwaysEvt(n))
                          | Res i => if n >= i
                                     then alwaysEvt(n-i)
                                   else neverEvt
                          | Rel j => ...)
         in serverLoop(sync(evt))
         end
   in (spawn(serverLoop(n));
       reqCh)
   end
```

Fig. 2. A simple server abstraction that uses transactional events.

```
fun query(server,replyCh)  =
  thenEvt(sendEvt(server,Query(replyCh))
          fn () => recvEvt(replyCh)

fun client(server,f) =
  let val replyCh = channel()
      val evt =  thenEvt(query(server,replyCh),
                         fn (n) => let val k = f(n)
                                   in sendEvt(server,Res(k))
                                   end
  in sync(evt)
  end
```

Fig. 3. A client that defines a protocol involving multiple communication events with a server.

resources based on this state. While this solution is disarmingly simple, it is unfortunately incorrect. This is because the transactional event that defines the server loop can commit only when the client's transaction does. But, the client requires *two* operations involving the server to be atomically executed, the first to perform the query, and the second to make the actual reservation. However, the server will not initiate the next iteration of its server loop until the query first performed by the client successfully commits; this transaction cannot commit until the client can successfully complete its second communication with the server. In other words, the communication parity mismatch between the server and client foils the construction of a complex atomic client-side protocol: the client requires two communications with the server for its transaction to be successful, whereas the server only expects a single interaction. Simply changing the client implementation so that the query and subsequent reservation do not execute within a transactional event would break obvious atomicity requirements. Alternatively, we could change the server code to accept two requests as part of its transactional event; this would allow the client protocol to succeed for this example, but would be a very brittle solution, since it not work for other kinds of protocols that initiate a different number of communication actions with the server.

An alternative is to modify the server by allowing it to accept an arbitrary number of requests as part of a given transaction (see Fig. 4).

```
fun server (n) =
  let val reqCh = channel()
      fun serverLoop(n) =
            let val evt =
                 thenEvt(recvEvt(reqCh),
                         fn (req) =>
                             wrap (case req of
                                     Query ch =>
                                        wrapEvt(sendEvt(ch,n)
                                                fn () => alwaysEvt(n))
                                   |  Res i => if n < i
                                                  then neverEvt
                                               else alwaysEvt(n-i)
                                   |  Rel j => ...),
                                fn (n) => serverLoop(n))
             in sync(evt)
             end
    in (spawn(serverLoop(n));
        reqCh)
    end
```

Fig. 4. A server implementation that accepts multiple requests as part of a given transaction.

In this revised implementation, the internal server loop itself is implemented as a transactional event. This enables the server to receive multiple requests as part of a complex client protocol. Unfortunately, there remain problems with this solution as well. The simpler problem is that the loop itself does not have a termination condition, and thus has no apparent commit point; this can be easily fixed by extending the interface to allow clients to notify the server when the client-side transaction is complete.

A more problematic issue is the loss of serializability due to the possibility the server may accept requests from *different* clients while participating in an ongoing transaction. Consider clients C_1 and C_2 both implementing the protocol described above. Suppose both issue queries to the server and receive the same result indicating the current server state. If client C_1 now succeeds in performing its reservation, C_2's subsequent computation of its desired reservation, which was based on a server state that is no longer accurate, is now incorrect. Although atomicity is preserved – none of the operations results in a `neverEvt` being yielded by the server, and thus all communication actions succeed – isolation is lost because C_1's effects are visible in the middle of C_2's transaction.

To provide a solution that permits multiple clients to concurrently communicate with the server without violating serializability guarantees, we introduce a new abstraction that *isolates* the execution of one transactional event from the concurrent effects performed by another. The abstraction is expressed using a new event combinator: `isolateEvt`: `'a evt` $\rightarrow$ `'a evt` that given an event value yields an isolated event value that when synchronized executes in isolation of all other isolated events.

Thus, to ensure that a client executes in isolation from all other clients, we could write:

```
fun client(server,f,dom) =
  let val replyCh = channel()
      val evt =  client protocol
  in sync(isolateEvt(evt))
  end
```

To prevent interference induced by such communication, isolate evaluation ensures that all communication to the server by C_1 are performed prior to all communication by C_2 or *vise versa*, effectively serializing their execution with respect to their common channels. Because the communication actions performed by the event arguments to an `isolateEvt` may be arbitrarily complex, enforcing such ordering requires tracking the effects performed by the event arguments transitively; in the following sections, we discuss how to efficiently identify and collect this information.

4 Semantics

Our semantics is defined in terms of a core call-by-value functional language with threading and communication primitives. Communication between threads is achieved using synchronous channels and transaction-encapsulated events. Our

language extends a transactional event core language with one additional construct used to express isolated events. For perspicuity, the language omits many useful event combinators such as `chooseEvt`, `wrap`, or `guard` since they raise no interesting semantic issues with respect to isolates. References are also omitted for this reason.

We first present a semantics for this language whose syntax and grammar is shown in Fig. 5 using a naive definition of isolates in which concurrent evaluation of isolated events is not allowed; we subsequently consider refinements to this semantics that relax this restriction.

$$
\begin{array}{l}
e := \texttt{unit} \mid \gamma \mid x \mid \lambda x.e \mid e\ e \\
\quad \mid \texttt{spawn}\ e \mid \texttt{sync}\ e \mid \texttt{ch}() \\
\quad \mid \texttt{sendEvt}(e,e) \mid \texttt{recvEvt}(e) \\
\quad \mid \texttt{neverEvt} \mid \texttt{alwaysEvt}\ e \\
\quad \mid \texttt{thenEvt}\ (e,e) \mid \texttt{isolateEvt}(e)
\end{array}
\qquad
\begin{array}{l}
v := \texttt{unit} \mid c \mid \gamma \mid \lambda x.e \\
\quad \mid \texttt{sendEvt}(v,v) \mid \texttt{recvEvt}(v) \\
\quad \mid \texttt{neverEvt} \mid \texttt{alwaysEvt}\ v \\
\quad \mid \texttt{thenEvt}\ (v,v) \mid \texttt{isolateEvt}(v)
\end{array}
$$

$$
\begin{array}{l}
E := \cdot \mid E\ e \mid v\ E \\
\quad \mid \texttt{sync}\ E \\
\quad \mid \texttt{sendEvt}(E,e) \mid \texttt{sendEvt}(c,E) \\
\quad \mid \texttt{recvEvt}(E) \mid \texttt{alwaysEvt}\ E \\
\quad \mid \texttt{thenEvt}\ (E,e) \mid \texttt{thenEvt}\ (v,E) \\
\quad \mid \texttt{isolateEvt}(E)
\end{array}
\qquad
\begin{array}{l}
\mathcal{F} := \cdot \mid \\
\qquad \texttt{thenEvt}\ (\mathcal{F},v) \mid \\
\qquad \texttt{isolateEvt}(v) \mid \\
\qquad \texttt{alwaysEvt}\ v
\end{array}
$$

$$
\begin{array}{rl}
M \in \mathit{ContextStack} & := \mathbf{I} \mid E \mid \mathcal{F} \mid M : M \\
T \in \mathit{NonTransactionThread} & := (\texttt{t}, e) \\
\overline{T} & := T \mid T \mid\mid \overline{T} \\
K \in \mathit{TransactionThread} & := (\texttt{t}, M, e) \\
\overline{K} & := K \mid K \mid\mid \overline{K}
\end{array}
$$

Fig. 5. Language Syntax and Grammar

In our syntax (see Fig. 5) v ranges over values, c over channel references, γ over constants, e over expressions, and $\texttt{t}$ over thread identifiers. The semantics shown in Fig. 6 defines three relations. The $\hookrightarrow$ relation defines thread-local actions that can be performed within or outside a transactional context. Function application (rule App) and channel creation (rule Channel) can be performed in either context.

Global evaluation is defined via relation $\rightarrow$. A global state consists of a set of transactional threads ($\overline{K}$) and non-transactional threads ($\overline{T}$). Threads that get spawned (rule Spawn) are initially not part of any ongoing transaction, and are initially added to the set of non-transactional threads ($\overline{T}$). The StepThread rule simply allows local execution within a non-transactional thread.

A thread gets added to $\overline{K}$ when it attempts to synchronize an event (rule SyncThread). Thus, *every* event synchronization initiates a transaction. For

App

$$\frac{}{(\lambda x.e)v \hookrightarrow e[v/x]}$$

Channel

$$\frac{c \ \mathit{fresh}}{\texttt{ch}() \hookrightarrow c}$$

Spawn

$$\frac{\texttt{t}' \ \mathit{fresh}}{\langle \overline{K}, (\texttt{t}, E[\texttt{spawn}\ e]) \mid\mid \overline{T}\rangle \rightarrow \langle \overline{K}, (\texttt{t}', [e]) \mid\mid (\texttt{t}, E[\texttt{unit}]) \mid\mid \overline{T}\rangle}$$

SyncThread

$$\frac{}{\langle \overline{K}, (\texttt{t}, E[\texttt{sync}\ v])\rangle \mid\mid \overline{T} \rightarrow \langle (\texttt{t}, E, v) \mid\mid \overline{K}, \overline{T}\rangle}$$

StepThread

$$\frac{e \hookrightarrow e'}{\langle \overline{K}, (\texttt{t}, E[e]) \| \overline{T}\rangle \rightarrow \langle \overline{K}, (\texttt{t}, E[e']) \| \overline{T}\rangle}$$

StepTransactionalThread

$$\frac{\overline{K} \rightsquigarrow \overline{K}'}{\langle \overline{K}, \overline{T}\rangle \rightarrow \langle \overline{K}', \overline{T}\rangle}$$

CommitTransThreads

$$\frac{\overline{K} = (\texttt{t}_1, E_1, \texttt{alwaysEvt}\ v_1) \mid\mid ... \mid\mid (\texttt{t}_n, E_n, \texttt{alwaysEvt}\ v_n) \quad \overline{T}' = \overline{T} \mid\mid (\texttt{t}_1, E_1[v_1]) \mid\mid ... \mid\mid (\texttt{t}_n, E_n[v_n])}{\langle \overline{K}, \overline{T}\rangle \rightarrow \langle \phi, \overline{T}'\rangle}$$

StepRunThread

$$\frac{e \hookrightarrow e'}{\overline{K} \mid\mid (\texttt{t}, M, \mathcal{F}[e]) \rightsquigarrow \overline{K} \mid\mid (\texttt{t}, M, \mathcal{F}[e'])}$$

NestedSync

$$\frac{}{\overline{K} \mid\mid (\texttt{t}, M, \mathcal{F}[\texttt{sync}\ v]) \rightsquigarrow \overline{K} \mid\mid (\texttt{t}, \mathcal{F} : M, v)}$$

NestedSyncComplete

$$\frac{}{\overline{K} \mid\mid (\texttt{t}, \mathcal{F} : M, \texttt{alwaysEvt}\ v) \rightsquigarrow \overline{K} \mid\mid (\texttt{t}, M, \mathcal{F}[v])}$$

ThenAlways

$$\frac{}{\overline{K} \mid\mid (\texttt{t}, M, \mathcal{F}[\texttt{thenEvt}(\texttt{alwaysEvt}\ (v_1), v_2)]) \rightsquigarrow \overline{K} \mid\mid (\texttt{t}, M, \mathcal{F}[v_2\ v_1])}$$

SendRecv

$$\frac{}{(\texttt{t}_1, M_1, \mathcal{F}_1[\texttt{sendEvt}(c, v)]) \mid\mid (\texttt{t}_2, M_2, \mathcal{F}_2[\texttt{recvEvt}(c)]) \mid\mid \overline{K} \rightsquigarrow (\texttt{t}_1, M_1, \mathcal{F}_1[\texttt{alwaysEvt unit}]) \mid\mid (\texttt{t}_2, M_2, \mathcal{F}_2[\texttt{alwaysEvt}\ v]) \mid\mid \overline{K}}$$

IsolateEvt

$$\frac{(\texttt{t}_1, M_1 : \mathbf{I} : M_2, e) \notin \overline{K}}{\overline{K} \mid\mid (\texttt{t}, M, \mathcal{F}[\texttt{isolateEvt}(v)]) \rightsquigarrow \overline{K} \mid\mid (\texttt{t}, \mathbf{I} : M, \mathcal{F}[v])}$$

NestedIsolateEvt

$$\frac{}{\overline{K} \mid\mid (\texttt{t}, M_1 : \mathbf{I} : M_2, \mathcal{F}[\texttt{isolateEvt}(v)]) \rightsquigarrow \overline{K} \mid\mid (\texttt{t}, M_1 : \mathbf{I} : M_2, \mathcal{F}[v])}$$

IsolateEvtComplete

$$\frac{}{\overline{K} \mid\mid (\texttt{t}, \mathbf{I} : M, \texttt{alwaysEvt}\ v) \rightsquigarrow \overline{K} \mid\mid (\texttt{t}, M, \texttt{alwaysEvt}\ v)}$$

Fig. 6. High-level semantics.

our purposes here, complex events are only built using `thenEvt`s. A thread executing transactionally is represented as a triple, $(\mathtt{t}, M, v)$ consisting of the thread identifier, a context stack that holds the continuation of the synchronization action, and the event. Rule StepTransactionalThread yields new global states based on the evaluation of expressions within transactions.

If all transactional threads in $\overline{K}$ have completed (i.e., have evaluated the event that they are synchronized upon to an `alwaysEvt`(v)), they can be removed from the transactional thread set and may resume execution as regular non-transactional threads (rule CommitTransThreads). The requirement that *all* threads complete is essential for ensuring atomicity in the presence of communicating message-passing operations. The value encapsulated by the `alwaysEvt` combinator fills the context that surrounded the original `sync` operation that made the computation transactional. Recall that the context is recorded in a *context stack*. Contexts must be saved on a stack because transactions can be nested, as we describe below. When the stack contains a single entry, and the thread term is of the form `alwaysEvt`(v), the thread is guaranteed to be no longer executing transactionally.

Transactional evaluation is defined via relation $\rightsquigarrow$ and takes place within $\mathcal{F}$ evaluation contexts; these contexts enforce atomicity of transactional execution by preventing intermediate actions from being visible to other non-transactional threads until the transaction fully completes.

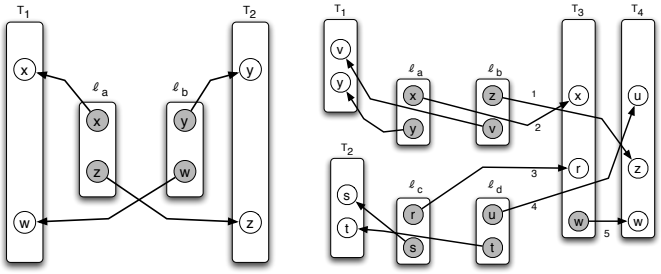
The StepRunThread rule allows local reductions within transactions. Rule NestedSync permits new transactions (initiated by a `sync` action) to be instantiated within existing ones. The current context of the outer transaction is recorded on the context transaction stack. When a transaction completes (rule NestedSyncComplete), the saved context of the outer transaction is popped from the stack and the value yielded by the completed inner transaction is supplied to fill its hole. Since transactions can be nested, the stack is necessary to record the distinction between nested and top-level transactions, to facilitate the transition from transactional to non-transactional execution. Note that a transactional event that is synchronized upon can only successfully commit if it yields an `alwaysEvt` value – an expression that yields `neverEvt` can never be successfully synchronized.

A `thenEvt` evaluates its first event argument to an `alwaysEvt` value containing argument v_1, and applies the function defined by its second argument (v_2) to v_1 (rule ThenAlways). Two transactional threads can communicate via a `sendEvt` and `recvEvt`.

Rule IsolateEvt defines isolate evaluation. An isolate combinator given an event value enforces isolation on that event value by prohibiting concurrent evaluation of any other isolates; we record that a transactional thread is executing an isolate by explicitly marking the context stack using token **I**. Rule NestedIsolateEvt allows a thread executing an isolate to initate another one. When an isolate completes, the thread state reverts to an ordinary transaction (rule IsolateEvtComplete).

5 Concurrent Isolate Execution

The formulation of isolates given in the previous section enforces serializability by preventing threads executing different isolates from executing concurrently. Unfortuately, simply removing this constraint to extract greater concurrency can lead to incorrect (non-isolated) executions. Fig.7(a) depicts one such example. Here, isolate ℓ_a initiates communication with T_1 (and then T_2); similarly, isolate ℓ_b initiates a synchronous communication action with T_2 and then T_1. The resulting global state could *not* have been produced via a serial execution of the two isolate computations – if ℓ_a were sequenced before ℓ_b (or *vise versa*) the communication with thread T_2 (or conversely, T_1) would be blocked.



(a) (b)

Fig. 7. Serializability violations introduced by incorrect parallel evaluation of isolates.

Even when isolate computations are serialized with respect to a single thread, care must still be taken to ensure *all* threads witness a consistent serializable view of the different isolates they communicate with. Consider the execution depicted in Fig. 7(b). Observe that T_1 witnesses an ordering in which ℓ_a executes before ℓ_b. Similarly, T_2 observes a serialization in which ℓ_c occurs before ℓ_d. Each of the isolates shown in the figure also communicate with T_3 and T_4. Thread T_3 participates in an action with ℓ_a before an action from ℓ_c; T_4 participates in an action with ℓ_d before ℓ_b; the communication between T_3 and T_4 (labelled edge 5) must enforce this ordering. There are now two serializable executions induced by these different actions. From T_1 and T_4's perspective, we need to ensure that ℓ_d, ℓ_b, ℓ_a and ℓ_c are executed atomically in that order. On the other hand, from T_2 and T_3's perspective, the required ordering is ℓ_a, ℓ_c, ℓ_d and ℓ_b.

Clearly, there is no schedule that satisfies both evaluation order sequences, even though each thread on its own witnesses what appears to be a consistent serializable execution. The inconsistency occurs because every thread must share the same view of how isolates are serialized with respect to one another.

5.1 Capabilities

To enable scalable construction of such views, we formulate a new semantics that associates *capabilities* with threads and isolates. Capabilities are used to indicate the isolate computations a given thread may communicate with. We prevent the global serializability failure of the previous example by tracking *serialization order* globally; i.e. when an isolate discovers a potential serialization order based on how other isolates have communicated, subsequent communications that violate this order are prohibited. As such, a communication action that forces a serialization order can be viewed as a *commit point* for that isolate.

A capability is defined as a mapping between labels ℓ denoting the dynamic instances of isolate event expressions, and *tags*. When an isolate event ℓ is synchronized, a constraint is established that relates the execution of the thread executing this isolate with threads evaluating other isolates. This constraint is modeled by the tag. Intuitively, from ℓ's perspective, the set of all isolates can be partitioned into two sets: the set `L` that only includes itself, and the set `R` that is composed of I, the set of all other evaluating isolates. To ensure a thread T's communication with ℓ is serializable, it must be the case that either (a) T has thus far communicated only with ℓ (i.e., the sole element in set `L`); (b) T has thus far communicated only with isolates other than ℓ (i.e., elements in the set `R`); or (c) T had previously communicated with isolates in I, but now only communicates with ℓ. One possibility is prohibited: T cannot have previously communicated with ℓ, and then subsequently engaged in communication with isolates in I; allowing T to engage in further communications with ℓ would break obvious isolation guarantees on ℓ's execution.

Thus, suppose thread T has a capability $\ell \mapsto$ `L`. The tag `L` indicates that the thread witnesses only the actions performed by ℓ, and no other isolate computation. Conversely, if T is given capability $\ell \mapsto$ `R`, it means it has only communicated with isolates other than ℓ. Tag `LR` indicates that T has first communicated with ℓ and then other isolates; tag `RL` indicates that T has first communicated with other isolates before communicating exclusively with ℓ.

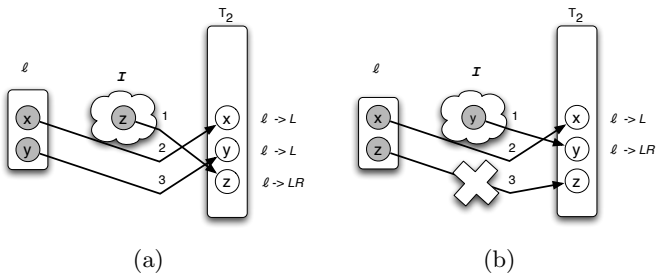


Fig. 8. Capabilities regulate communication with isolates.

Examples Consider the two examples given in Fig. 8. In both diagrams, there are two isolate expressions ℓ and I, resp. The expression I represents an abstract set of isolates. Consider the execution shown in Fig. **??**. ℓ performs two communication actions with thread T; once these actions complete some isolate from the set I initiates a communication as well. After the communication with ℓ, T's capability map has the entry: $[\ell \mapsto LR]$. The entry indicates that T first engaged in a communication with ℓ and then with other isolates found in I.

Now, consider the more complex example shown in Fig. 8(a)). Isolate ℓ attempts to perform two communications with thread T. Between these communications is a communication action performed by another isolate in I. The first communication by ℓ results in T acquiring a capability map containing capabilities $\ell \mapsto L$. The first capability indicates that T has only communicated with ℓ. The capability map is adjusted after the second communication via channel `y`. Now the capability map contains the capability $\ell \mapsto LR$. When ℓ tries to communicate via `z`, a serializability violation occurs: the capability $\ell \mapsto LR$ indicates that T has previously communicated with ℓ and then some other isolate in I, in that order. Allowing the communication event on `z` to occur would thus violate our required serializability invariant since ℓ's actions with respect to T cannot be grouped in their entirety either before or after I's.

5.2 Semantics

We define the semantics in Figs. 9 and 13. The semantics makes use of a new relation $\rhd$ (see Tag Enrichment) that captures the obvious ordering relationship among tags. Informally, tag ι can be enriched to tag ι' if ι imposes the same or fewer restrictions on isolate communication than ι'. Note the presence of two additional tags $\hat{L}$ and $\hat{R}$ that we haven't discussed thus far – these tags are used to build capabilites for threads executing isolate computations. If thread T executes isolate expression with label ℓ, its capability map is extended with the capability $\ell \mapsto \hat{L}$: such a capability can never be enriched since isolates, by definition, cannot communicate with other isolates. Similarly, the capability map of all other threads executing isolate computations is updated with the capability $\ell \mapsto \hat{R}$: such a capability prohibits these threads from communicating with ℓ.

Global evaluation is defined via relation $\longmapsto$ that is the analog of $\rightarrow$ in the semantics given in Fig. **??**. Transactional evaluation specifically (defined by relation $\Longrightarrow$) is now defined with respect to both global (Δ) and local (δ) capability maps. The global map fixes a specific serialization order for all "committed" isolates; the local map captures a thread's specific view of the isolates it has communicated with thus far prior to a commit point.

Rule Capability Enrichment introduces capabilities to a thread's capability map. If the thread does not already have a tag for this isolate, it can choose one (either `L` or `R`). The thread's capability map is then updated (via auxiliary function lift) to reflect this new binding. Rule Isolate Serialization allows enriching a capability to a serial ordering, namely `LR` or `RL`. This rule affects the global capability map. The antecedent condition $\iota \rhd \iota'$ leverages the structure of the relation to allow a capability that is currently `L` or `R` to become

$$\ell \in \mathit{Label}$$
$$\iota \in \mathit{Tag} := \cdot \mid \hat{\mathtt{L}} \mid \hat{\mathtt{R}} \mid \mathtt{L} \mid \mathtt{R} \mid \mathtt{LR} \mid \mathtt{RL}$$
$$\Delta \in \mathit{CapabilityMap} := \mathit{Label} \xrightarrow{\text{fin}} \mathit{Tag}$$
$$\delta \in \mathit{LocalCapabilityMap} := \mathit{TID} \xrightarrow{\text{fin}} \mathit{CapabilityMap}$$

Tag Enrichment

$$\begin{array}{cc} \cdot \rhd \mathtt{L} & \cdot \rhd \mathtt{R} \\ \mathtt{L} \rhd \mathtt{RL} & \mathtt{L} \rhd \mathtt{LR} \\ \mathtt{R} \rhd \mathtt{LR} & \mathtt{R} \rhd \mathtt{RL} \end{array}$$

Capability Lifting

$$\mathsf{lift}\ \ell\ \iota\ \delta\ \mathtt{t} = \delta[\mathtt{t} \mapsto\ \delta(\mathtt{t})[\ell \mapsto \iota]]$$

Capability Enrichment

$$\frac{\ell \notin \mathit{Dom}(\delta(\mathtt{t})) \qquad \iota' \in \{\mathtt{L},\mathtt{R}\} \qquad \delta' = \mathsf{lift}\ \ell\ \iota'\ \delta\ \mathtt{t}}{\Delta, \delta, \overline{K} \mid\mid (\mathtt{t}, M, e) \Longrightarrow \Delta, \delta', \overline{K} \mid\mid (\mathtt{t}, M, e)}$$

Isolate Serialization

$$\frac{\ell \notin \mathit{Dom}(\Delta) \qquad \delta\ \mathtt{t}\ \ell = \iota \qquad \iota \rhd \iota' \qquad \delta' = \mathsf{lift}\ \ell\ \iota'\ \delta\ \mathtt{t}}{\Delta, \delta, \overline{K} \mid\mid (\mathtt{t}, M, e) \Longrightarrow \Delta[\ell \mapsto \iota'], \delta', \overline{K} \mid\mid (\mathtt{t}, M, e)}$$

Serialization Check

$$\frac{\Delta(\ell) = \iota' \qquad \delta\ \mathtt{t}\ \ell = \iota \qquad \iota \rhd \iota' \qquad \delta' = \mathsf{lift}\ \ell\ \iota'\ \delta\ \mathtt{t}}{\Delta, \delta, \overline{K} \mid\mid (\mathtt{t}, M, e) \Longrightarrow \Delta, \delta', \overline{K} \mid\mid (\mathtt{t}, M, e)}$$

Fig. 9. Capabilities and Capability Enrichment.

LR or RL. Thus, this rule enforces a specific serialization order on an isolate and serves as a commit point for isolate evaluation. Since the global capability map defines a consistent serial ordering of isolates, all threads must adhere to this ordering. Rule Serialization Check thus "imports" the capability tag of an isolate found in the global state to a thread's local capability map.

Fig. 13 presents the salient rules that define transactional evaluation. Rule SyncThread begins transactional evaluation by initializing the local capability map of the thread performing the synchronization to the empty set, and adds a new transactional thread to $\overline{K}$. Conversely, rule CommitTransThreads reverts the state from transactional execution to ordinary evaluation by "clearing" the global and local capability maps. The appendix provides a complete definition of all the rules.

Rule IsolateFresh commences an isolate computation for a transactional event that has not yet communicated with any other isolate. A new isolate label (ℓ) is created, and the local capability map of the thread t performing the operation is updated to reflect this fact; δ' is the capability map that binds ℓ to L for t. To prevent serializability violations, we also require that all other threads ($\overline{\mathtt{t}'}$) currently executing isolate operations not communicate with this isolate. To facilitate this, we update the capability maps of these threads (δ'') to bind ℓ to R; note that we leverage the curried definition of lift in defining the fold. Isolation is enforced by the SendRecv rule shown below that prevents

SyncThread

$$\frac{}{\langle \Delta, \delta, \overline{K}, (\mathtt{t}, E[\mathtt{sync}\ v]) \mid\mid \overline{T}\rangle \longmapsto \langle \Delta, \delta[\mathtt{t} \mapsto \phi], (\mathtt{t}, E :\cdot, v) \mid\mid \overline{K}, \overline{T}\rangle}$$

CommitTransThreads

$$\frac{\overline{K} = (\mathtt{t}_1, E_1, \mathtt{alwaysEvt}\ v_1) \mid\mid ... \mid\mid (\mathtt{t}_n, E_n, \mathtt{alwaysEvt}\ v_n) \quad \overline{T}' = \overline{T} \mid\mid (\mathtt{t}_1, E_1[v_1]) \mid\mid ... \mid\mid (\mathtt{t}_n, E_n[v_n])}{\langle \Delta, \delta, \overline{K}, \overline{T}\rangle \longmapsto \langle \phi, \phi, \phi, \overline{T}'\rangle}$$

IsolateFresh

$$\frac{\ell\ \mathit{fresh}\ \ \delta(\mathtt{t}) = \phi\ \ \delta' = \mathsf{lift}\ \ell\ \hat{\mathtt{L}}\ \delta\ \mathtt{t}\ \ \bar{\mathtt{t}}' = \{\mathtt{t} \mid (\mathtt{t}, M_1 : \mathbf{I} : M_2, e') \in \overline{K}\ \ \delta'' = \mathsf{fold}(\mathsf{lift}\ \ell\ \hat{\mathtt{R}},\ \delta',\ \bar{\mathtt{t}}')}{\Delta, \delta, \overline{K} \mid\mid (\mathtt{t}, M, F[\mathtt{isolateEvt}(v)]) \Longrightarrow \Delta, \delta'', \overline{K} \mid\mid (\mathtt{t}, \mathbf{I} : M, F[\mathtt{v}])}$$

IsolateComm

$$\frac{\ell\ \mathit{fresh}\ \ \delta(\mathtt{t}) \neq \phi\ \ \delta' = \mathsf{lift}\ \ell\ \mathtt{RL}\ \delta\ \mathtt{t}\ \ \bar{\mathtt{t}}' = \{\mathtt{t} \mid (\mathtt{t}, M_1 : \mathbf{I} : M_2, e') \in \overline{K}\ \ \delta'' = \mathsf{fold}(\mathsf{lift}\ \ell\ \mathtt{RL},\ \delta',\ \bar{\mathtt{t}}')}{\Delta, \delta, \overline{K} \mid\mid (\mathtt{t}, M, F[\mathtt{isolateEvt}(v)]) \Longrightarrow \Delta, \delta'', \overline{K} \mid\mid (\mathtt{t}, \mathbf{I} : M, F[\mathtt{v}])}$$

SendRecv

$$\frac{\Delta' = \delta(\mathtt{t}_1) \diamond \delta(\mathtt{t}_2)}{\Delta, \delta, (\mathtt{t}_1, M_1, F_1[\mathtt{sendEvt}(c, v)]) \mid\mid (\mathtt{t}_2, M_2, F_2[\mathtt{recvEvt}(c)]) \mid\mid \overline{K} \Longrightarrow \Delta, \delta[\mathtt{t}_1 \mapsto \Delta', \mathtt{t}_2 \mapsto \Delta'], (\mathtt{t}_1, M_1, F_1[\mathtt{alwaysEvt\ unit}]) \mid\mid (\mathtt{t}_2, M_2, F_2[\mathtt{alwaysEvt}\ v]) \mid\mid \overline{K}}$$

Fig. 10. Concurrent isolate evaluation using capabilities.

communication between threads that have either $\hat{\mathtt{L}}$ and $\hat{\mathtt{R}}$ bindings for the same isolate label. Thus, for a given isolate's lifetime, it cannot communicate with any other isolate either currently executing or which commences evaluation at some later point. However, isolates are still free to communicate with other threads not themselves executing isolate computations provided that such communications are permissible as defined by Fig. **??**.

Rule IsolateComm handles the case when a thread executing a transactional event which has previously communicated with other isolates, commences execution of a new isolate (i.e., the thread has capability bindings for other isolates in its capability map). In this case, the newly created isolate (call it I) should be serialized *after* the isolate the transaction has previously communicated with. If it were serialized before, it would imply the isolate was created prior to the communication which is obviously incorrect. The rule, therefore, chooses the appropriate serial ordering ($\mathtt{RL}$) and propagates it to all active isolates; rule IsolateSerialization takes care of updating the global capability map with this binding when any non-isolated thread communicates with an isolate event.

Rule SendRecv allows two threads to communicate provided their capability maps are *compatible*. Informally, two maps are compatible if the tags for the isolates they have in common are the same. More precisely, $I \diamond I'$ if $Dom(I) = Dom(I')$ and$I(\ell) \diamond I(\ell')$ where $\hat{\mathtt{L}} \diamond \mathtt{L}$, $\hat{\mathtt{L}} \diamond \mathtt{RL}$, $\hat{\mathtt{R}} \diamond \mathtt{R}$, $\hat{\mathtt{R}} \diamond \mathtt{LR}$ and $\iota \diamond \iota$. For example, $\hat{\mathtt{L}} \diamond \mathtt{L}$ holds because a thread executing an isolate ℓ (thus having capability $\ell \mapsto \hat{\mathtt{L}}$

can certainly communicate with a thread that has previously only communicated with this isolate (that thread would therefore have capability $\ell \mapsto \texttt{L}$). As another example, if thread T_1 had previously communicated with isolate ℓ and thread T_2 had previously communicated with isolate ℓ', T_1 must acquire the capability for ℓ' maintained by T_2 and T_2 must acquire the capability for ℓ maintained by T_1. Capability enrichment allows this to happen, and thus prevents T_1 from subsequently communicating with ℓ' in ways inconsistent with tag associated with T_2's capability on ℓ'. In other words, to preserve isolation and enforce serializability, we must ensure that effects from one isolate argument does not leak into another via third-party interaction.

Notice that not none of the rules in Fig. **??** update the global capability map Δ. Consistency among the capabilities two communicating threads have is enforced by the $\diamond$ relation in rule SendRecv that is satisfied via the capability enrichment rules.

Examples The first example shown in Fig. 11(a) is a refined version of Fig. 7(a). The isolate ℓ_a is created initially. Since there are no other isolates, it is mapped to $\hat{\texttt{L}}$ in the capability map for the thread that executes it. Subsequently, isolate ℓ_b is created. The creation of the second isolate adds the capability $\ell_b \mapsto \hat{\texttt{R}}$ to ℓ_a's thread's capability map. Now, when thread `T` communicates with isolates ℓ_a it inherites ℓ_a's bindings. No new bindings are introduced on the second communication with ℓ_a, but when `T` communicates to ℓ_b, `T`'s capability for ℓ_b is lifted to `RL`.

The second example, shown in Fig. 11(b), illustrates capability enrichment in the case when two threads have communicated with two seperate isolates. Like the previous example, after the creation of ℓ_b, the thread executing ℓ_a has a local capability map with capabilities $\ell_a \mapsto \hat{\texttt{L}}$ and $\ell_b \mapsto \hat{\texttt{R}}$. When ℓ_a communicates with T_1, T_1 acquires capabilities $\ell_a \mapsto \texttt{L}$ and $\ell_b \mapsto \texttt{R}$. Similarly, when T_2 communicates with isolate ℓ_b its capability map becomes $\ell_b \mapsto \texttt{R}$. When T_1 and T_2 communicate they must coordinate to make sure they both see a consistent view of the isolates they have communicated with. Both threads agree on ℓ_a as T_2 can lift its capability for ℓ_a to `L`. However, for ℓ_b, T_1 has a capability `R` and T_2 has a capability `L`. In this case, either serial ordering of `LR` or `RL` is possible as the threads have never interacted before. Therefore, we simply choose a particular ordering, here `RL`. After this point, neither T_1 nor T_2 can communicate with ℓ_a – the serial ordering defined by the capability map dictates that communication with ℓ_a precede all communication with ℓ_b.

6 Soundness

Our main soundness result asserts that concurrent isolate evaluation (as defined by Figs. 9 and 13) is equivalent to serial isolate evaluation (as defined by Fig. 6):

Theorem. *If*
$\Delta, \delta, \overline{K} \mid\mid (\texttt{t}, M, \mathcal{F}[\texttt{isolateEvt}(v)]) \Longrightarrow \ldots \Longrightarrow \Delta', \delta', \overline{K}' \mid\mid (\texttt{t}, \mathbf{I} : M, \mathcal{F}[\texttt{alwaysEvt}(v')])$

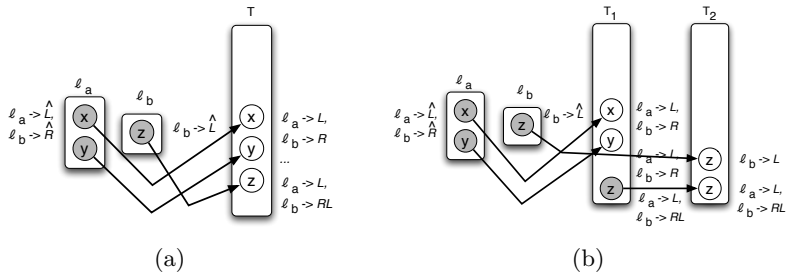


Fig. 11. Capabilities permit threads to communicate with multiple isolates, even those that are nested within one another, provided serializability invariants are preserved.

then
$\overline{K} \mid\mid (\mathtt{t}, M, \mathcal{F}[\mathtt{isolateEvt}(v)])) \rightsquigarrow \ldots \rightsquigarrow \overline{K}' \mid\mid (\mathtt{t}, \mathbf{I} : M, \mathcal{F}[\mathtt{alwaysEvt}(v')])$

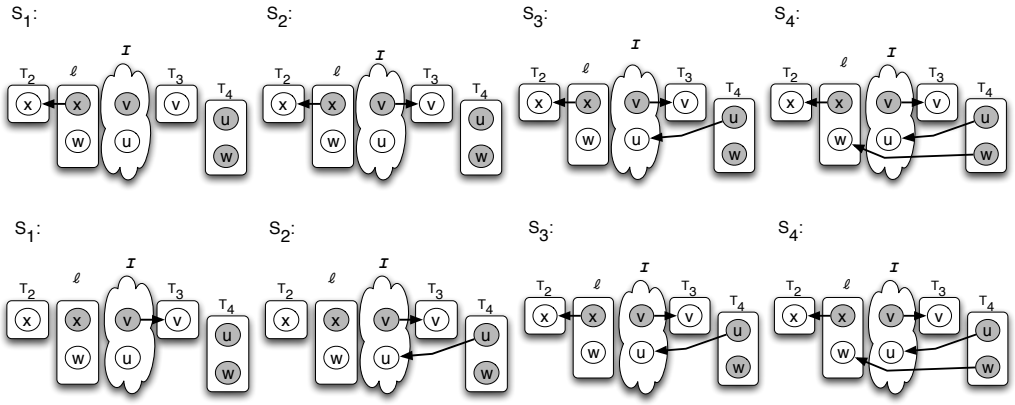


Fig. 12. Serializability of concurrent evaluation means all actions performed by the left-hand side of an isolate can be permuted before the actions performed by the right and *vise versa*.

The intuition underlying the proof for Theorem **??** is shown in Fig. 12. States S_1 through S_4 depict communication between an isolate with label ℓ and a set of other isolates I with threads T_2, T_3 and T_4. In S_1, isolate ℓ communicates with T_2 via channel `x`; in S_2, I communicates with T_3 via channel `v`; in S_3, there is a communication between T_4 and I; and, finally in S_4, T_4 establishes a communication with ℓ. Even though the isolate ℓ and the set of isolates I interleave their communication actions, their behavior is equivalent to an execution in which, for example, the I fully completes before ℓ. This alternative schedule is shown via states S_1' through S_4' in the figure. Here, each of the right-hand side

actions are performed before the left. Note that S_4 and S'_4 are identical. It is this permutability property on communication actions enforced by isolates that ensures our soundness result.

7 Implementation

There are three critical issues in building an efficient and feasible implementation of transactional events and isolation: (1) performing a systematic search of potential communication actions, initiating exploration of new communication pairings when an existing search yields `neverEvt`, or fails to make progress (e.g., deadlocks); (2) determining when to perform capability enrichment on local capability maps; and, (3) minimizing the cost of maintaining and updating the global map.

When a thread commences evaluation of an isolate executing within a transactional event, a new *search thread* is created (5; 6) that attempts to discharge all communication actions in that event that preserves serializability with respect to all other isolate computations. A serializability violation may manifest either because an event synchronization yields `neverEvt`, or a violation of isolation is discovered (i.e., the $\diamond$ relation fails to hold on a communication). In either case, we must abort the search thread, and revert any effects it has induced. To do this, we leverage *stabilizers* (24), a lightweight checkpointing mechanism meant to provide global state recovery for CML. Once a particular communication sequence is aborted, a new search is undertaken to explore an alternative set of communication pairings. Stabilizers provide sufficient information to synthesize a new search space from an aborted one.

We can restrict when capability enrichment on local capability maps occur to communication actions. Instead of allowing threads to lift their capabilities at arbitrary points *prior* to a communication, our implementation lifts capabilities *during* a communication. If one thread does not have a capability binding, it simply inherits one from its communicaiton partner. If both threads contain a capability mapping then they must agree or communication is not possible. We represent capabilities using two bits of information communicated along with the message. If agreement results in a new serialization (i.e., lifting to `LR` or `RL` when neither of the communicating threads was `LR` or `RL` prior to the communication), we check the global capability map for this isolate. If there was no previous serialization of this isolate, we commit this serialization; if one existed, we check to see if the two are equal. If the new serialization does not match the old, the communication is invalid.

The global capability map needs to be consulted at most once per thread precisely at a communication. When two threads engage in a communication that must agree on a particular serialization, they consult their local capability maps. Once agreement is reached, the global capability map must be updated only if a particular serial order (i.e., `LR` or `RL`) for an isolate is required. The global capability prevents any other serialization of such isolates. Once such an

order is decided, the thread never needs to subsequently consult the global map for that isolate since the tag enrichment relation does not lift `LR` or `RL` tags.

8 Related Work and Conclusions

Our work fully integrates a transactional semantics into the CML, a concurrent extension to SML that supports synchronous message passing (17). Previous work has leveraged the atomicity guarantee of transactional semantics to add expressivity to message-passing abstractions by introducing transactional events (5; 6). These proposals allow multiple communicating actions encapsulated by transactional events to make their effects visible only once all communicating actions can succeed. However, they do not provide a means to restrict the visibility of one transactional event from another, and therefore do not guarantee serializability of transactional computations. In (6), the authors extended transactional events to support mutable references and nested synchronizations. Because this formulation of transactional events follows (5), nested transactions can always be flattened so that they execute as part of a top-level outer transaction.

In contrast, the focus of our work is a formulation of transactional events that allows programmers to achieve isolation among concurrently executing transactional events via an isolate combinator. Guaranteeing isolation requires tracking the communication of all threads that have communicated with isolates, and ensuring the communications do not violate serializability. Furthermore, nested isolates cannot be flattened, because the effects performed by a nested isolate must remain isolated from all transactional computations specified at each nesting level.

Synchronous message-passing and event abstractions, like those in CML, have also been introduced in other languages. For example, an implementation of events for Concurrent Haskell has been described in (19), for Scheme in (7) and for Caml in (4). Asynchronous message-passing has been used in languages like Erlang (2). Transactional events (5; 6) have been implemented in Concurrent Haskell and CML, respectively.

Transactional memory systems were proposed in (13). Previous work (21; 9; 3; 12; 11; 16; 22; 1; 20) has considered using software transactions to simplify reasoning about interactions between concurrently executing computations in a shared-memory system. Implementations for software transactional memory have been presented in functional languages such as Caml (18), Scheme (14) and Haskell (10). For example, the implementation of transactional events presented in (5) uses a software transactional memory extension of Concurrent Haskell (10) available in the Glasgow Haskell Compiler (GHC) (8).

There has also been other previous work that has combined inter-thread communication and software transactions. In (23) the authors extend tradition transactional memory with the ability to observe the effects of other threads at selected points. (24) describes the construction of transparent checkpoints, which can be used for transactional roll-back, in the context of CML.

A Appendix

Lemma 1. *If*
$\langle\, \Delta, \delta, (\mathtt{t}_1, M_1, e_1)||(\mathtt{t}_2, M_2, e_2)||\overline{K}, \overline{T}\,\rangle \Longrightarrow$
$\langle\, \Delta, \delta, (\mathtt{t}_1, M_1, e_1)||(\mathtt{t}_2, M_2, e_2')||\overline{K}, \overline{T}\,\rangle \Longrightarrow$
$\langle\, \Delta, \delta, (\mathtt{t}_1, M_1, e_1')||(\mathtt{t}_2, M_2, e_2')||\overline{K}, \overline{T}\,\rangle$
then
$\langle\, \Delta, \delta, (\mathtt{t}_1, M_1, e_1)||(\mathtt{t}_2, M_2, e_2)||\overline{K}, \overline{T}\,\rangle \Longrightarrow$
$\langle\, \Delta, \delta, (\mathtt{t}_1, M_1, e_1')||(\mathtt{t}_2, M_2, e_2)||\overline{K}, \overline{T}\,\rangle \Longrightarrow$
$\langle\, \Delta, \delta, (\mathtt{t}_1, M_1, e_1')||(\mathtt{t}_2, M_2, e_2')||\overline{K}, \overline{T}\,\rangle$

Lemma 2. *If*
$\langle\, \Delta, \delta, \overline{K}, (\mathtt{t}_1, e_1)||(\mathtt{t}_2, e_2) \;||\; \overline{T}\,\rangle \longmapsto$
$\langle\, \Delta, \delta, \overline{K}, (\mathtt{t}_1, e_1')||(\mathtt{t}_2, e_2) \;||\; \overline{T}\,\rangle \longmapsto$
$\langle\, \Delta, \delta, \overline{K}, (\mathtt{t}_1, e_1')||(\mathtt{t}_2, e_2') \;||\; \overline{T}\,\rangle$
then
$\langle\, \Delta, \delta, \overline{K}, (\mathtt{t}_1, e_1)||(\mathtt{t}_2, e_2) \;||\; \overline{T}\,\rangle \longmapsto$
$\langle\, \Delta, \delta, \overline{K}, (\mathtt{t}_1, e_1)||(\mathtt{t}_2, e_2') \;||\; \overline{T}\,\rangle \longmapsto$
$\langle\, \Delta, \delta, \overline{K}, (\mathtt{t}_1, e_1')||(\mathtt{t}_2, e_2') \;||\; \overline{T}\,\rangle$

Lemma 3. *If*
$\langle\, \Delta, \delta, \overline{K} \;||\; (\mathtt{t}_1, M_1, e_1) \;||\; (\mathtt{t}_2, M_2, e_2), \overline{T}\,\rangle \Longrightarrow$
$\langle\, \Delta[\ell \mapsto \iota'], \delta', \overline{K} \;||\; (\mathtt{t}_1, M_1, e_1) \;||\; (\mathtt{t}_1, M_1, e_1), \overline{T}\,\rangle \Longrightarrow$
$\langle\, \Delta[\ell \mapsto \iota', \ell' \mapsto \iota''], \delta'', \overline{K} \;||\; (\mathtt{t}_1, M_1, e_1) \;||\; (\mathtt{t}_2, M_2, e_2), \overline{T}\,\rangle$
then
$\langle\, \Delta, \delta, \overline{K} \;||\; (\mathtt{t}_1, M_1, e_1) \;||\; (\mathtt{t}_2, M_2, e_2), \overline{T}\,\rangle \Longrightarrow$
$\langle\, \Delta[\ell' \mapsto \iota''], \delta''', \overline{K} \;||\; (\mathtt{t}_1, M_1, e_1) \;||\; (\mathtt{t}_2, M_2, e_2), \overline{T}\,\rangle \Longrightarrow$
$\langle\, \Delta[\ell \mapsto \iota', \ell' \mapsto \iota''], \delta'', \overline{K} \;||\; (\mathtt{t}_1, M_1, e_1) \;||\; (\mathtt{t}_2, M_2, e_2), \overline{T}\,\rangle$

Lemma 4. *If*
$\langle\, \Delta, \delta, (\mathtt{t}_1, M_1, e_1)||(\mathtt{t}_2, M_2, \mathcal{F}_2[\mathtt{sendEvt}(ch, v)])||(\mathtt{t}_3, M_3, \mathcal{F}_3[\mathtt{recvEvt}(ch)])||\overline{K}, \overline{T}\,\rangle \Longrightarrow$
$\langle\, \Delta, \delta, (\mathtt{t}_1, M_1, e_1)||(\mathtt{t}_2, M_2, \mathcal{F}_2[\mathtt{alwaysEvt\ unit}])||(\mathtt{t}_3, M_3, \mathcal{F}_3[\mathtt{alwaysEvt}\ v])||\overline{K}, \overline{T}\,\rangle \Longrightarrow$
$\langle\, \Delta, \delta, (\mathtt{t}_1, M_1, e_1')||(\mathtt{t}_2, M_2, \mathcal{F}_2[\mathtt{alwaysEvt\ unit}])||(\mathtt{t}_3, M_3, \mathcal{F}_3[\mathtt{alwaysEvt}\ v])||\overline{K}, \overline{T}\,\rangle$
then
$\langle\, \Delta, \delta, (\mathtt{t}_1, M_1, e_1)||(\mathtt{t}_2, M_2, \mathcal{F}_2[\mathtt{sendEvt}(ch, v)])||(\mathtt{t}_3, M_3, \mathcal{F}_3[\mathtt{recvEvt}(ch)])||\overline{K}, \overline{T}\,\rangle \Longrightarrow$
$\langle\, \Delta, \delta, (\mathtt{t}_1, M_1, e_1')||(\mathtt{t}_2, M_2, \mathcal{F}_2[\mathtt{sendEvt}(ch, v)])||(\mathtt{t}_3, M_3, \mathcal{F}_3[\mathtt{recvEvt}(ch)])||\overline{K}, \overline{T}\,\rangle \Longrightarrow$
$\langle\, \Delta, \delta, (\mathtt{t}_1, M_1, e_1')||(\mathtt{t}_2, M_2, \mathcal{F}_2[\mathtt{alwaysEvt\ unit}])||(\mathtt{t}_3, M_3, \mathcal{F}_3[\mathtt{alwaysEvt}\ v])||\overline{K}, \overline{T}\,\rangle$

Lemma 5. *If*
$\langle\, \Delta, \delta, (\mathtt{t}_1, M_1, \mathcal{F}_1[\mathtt{sendEvt}(ch_1, v_1)])||(\mathtt{t}_2, M_2, \mathcal{F}_2[\mathtt{recvEvt}(ch_1)])$
$\quad||(\mathtt{t}_3, M_3, \mathcal{F}_3[\mathtt{sendEvt}(ch_2, v_2)])||(\mathtt{t}_4, M_4, \mathcal{F}_4[\mathtt{recvEvt}(ch_2)])||\overline{K}, \overline{T}\,\rangle \Longrightarrow$
$\langle\, \Delta, \delta, (\mathtt{t}_1, M_1, \mathcal{F}_1[\mathtt{alwaysEvt\ unit}])||(\mathtt{t}_2, M_2, \mathcal{F}_2[\mathtt{alwaysEvt}\ v])$
$\quad||(\mathtt{t}_3, M_3, \mathcal{F}_3[\mathtt{sendEvt}(ch_2, v_2)])||(\mathtt{t}_4, M_4, \mathcal{F}_4[\mathtt{recvEvt}(ch_2)])||\overline{K}, \overline{T}\,\rangle \Longrightarrow$
$\langle\, \Delta, \delta, (\mathtt{t}_1, M_1, \mathcal{F}_1[\mathtt{alwaysEvt\ unit}])||(\mathtt{t}_2, M_2, \mathcal{F}_2[\mathtt{alwaysEvt}\ v])$
$\quad||(\mathtt{t}_3, M_3, \mathcal{F}_3[\mathtt{alwaysEvtunit}])||(\mathtt{t}_4, M_4, \mathcal{F}_4[\mathtt{alwaysEvt}v])||\overline{K}, \overline{T}\,\rangle$
then
$\langle\, \Delta, \delta, (\mathtt{t}_1, M_1, \mathcal{F}_1[\mathtt{sendEvt}(ch_1, v_1)])||(\mathtt{t}_2, M_2, \mathcal{F}_2[\mathtt{recvEvt}(ch_1)])$

Spawn

$$\frac{\mathtt{t}'\ \mathit{fresh}}{\langle \Delta, \delta, \overline{K}, (\mathtt{t}, E[\mathtt{spawn}\ e]) \mid\mid \overline{T}\rangle \longmapsto \langle \Delta, \delta, \overline{K}, (\mathtt{t}', [e]) \mid\mid (\mathtt{t}, E[\mathtt{unit}]) \mid\mid \overline{T}\rangle}$$

SyncThread

$$\frac{}{\langle \Delta, \delta, \overline{K}, (\mathtt{t}, E[\mathtt{sync}\ v]) \mid\mid \overline{T}\rangle \longmapsto \langle \Delta, \delta[\mathtt{t} \mapsto \phi], (\mathtt{t}, E :\cdot, v) \mid\mid \overline{K}, \overline{T}\rangle}$$

StepThread

$$\frac{e \hookrightarrow e'}{\langle \Delta, \delta, \overline{K}, (\mathtt{t}, E[e]) \| \overline{T}\rangle \longmapsto \langle \Delta, \delta, \overline{K}, (\mathtt{t}, E[e']) \| \overline{T}\rangle}$$

StepTransactionalThread

$$\frac{\Delta, \delta, \overline{K} \Longrightarrow \Delta', \delta', \overline{K}'}{\langle \Delta, \delta, \overline{K}, \overline{T}\rangle \longmapsto \langle \Delta', \delta', \overline{K}', \overline{T}\rangle}$$

CommitTransThreads

$$\frac{\overline{K} = (\mathtt{t}_1, E_1, \mathtt{alwaysEvt}\ v_1) \mid\mid ... \mid\mid (\mathtt{t}_n, E_n, \mathtt{alwaysEvt}\ v_n) \quad \overline{T}' = \overline{T} \mid\mid (\mathtt{t}_1, E_1[v_1]) \mid\mid ... \mid\mid (\mathtt{t}_n, E_n[v_n])}{\langle \Delta, \delta, \overline{K}, \overline{T}\rangle \longmapsto \langle \phi, \phi, \phi, \overline{T}'\rangle}$$

StepRunThread

$$\frac{e \hookrightarrow e'}{\Delta, \delta, \overline{K} \mid\mid (\mathtt{t}, M, \mathcal{F}[e]) \Longrightarrow \Delta, \delta, \overline{K} \mid\mid (\mathtt{t}, M, \mathcal{F}[e'])}$$

NestedSync

$$\frac{}{\Delta, \delta, \overline{K} \mid\mid (\mathtt{t}, M, \mathcal{F}[\mathtt{sync}\ v]) \Longrightarrow \Delta, \delta, \overline{K} \mid\mid (\mathtt{t}, \mathcal{F} : M, v)}$$

NestedSyncComplete

$$\frac{}{\Delta, \delta, \overline{K} \mid\mid (\mathtt{t}, \mathcal{F} : M, \mathtt{alwaysEvt}\ v) \Longrightarrow \Delta, \delta, \overline{K} \mid\mid (\mathtt{t}, M, \mathcal{F}[v])}$$

ThenAlways

$$\frac{}{\Delta, \delta, \overline{K} \mid\mid (\mathtt{t}, M, \mathcal{F}[\mathtt{thenEvt}(\mathtt{alwaysEvt}\ (v_1), v_2)]) \Longrightarrow \Delta, \delta, \overline{K} \mid\mid (\mathtt{t}, M, \mathcal{F}[v_2\ v_1])}$$

IsolateFresh

$$\frac{\ell\ \mathit{fresh}\ \ \delta(\mathtt{t}) = \phi\ \ \delta' = \mathsf{lift}\ \ell\ \hat{\mathtt{L}}\ \delta\ \mathtt{t} \quad \bar{\mathtt{t}'} = \{\mathtt{t} \mid (\mathtt{t}, M_1 : \mathbf{I} : M_2, e') \in \overline{K} \quad \delta'' = \mathsf{fold}(\mathsf{lift}\ \ell\ \hat{\mathtt{R}},\ \delta',\ \bar{\mathtt{t}'})}{\Delta, \delta, \overline{K} \mid\mid (\mathtt{t}, M, F[\mathtt{isolateEvt}(v)]) \Longrightarrow \Delta, \delta'', \overline{K} \mid\mid (\mathtt{t}, \mathbf{I} : M, F[v])}$$

IsolateComm

$$\frac{\ell\ \mathit{fresh}\ \ \delta(\mathtt{t}) \neq \phi\ \ \delta' = \mathsf{lift}\ \ell\ \mathtt{RL}\ \delta\ \mathtt{t} \quad \bar{\mathtt{t}'} = \{\mathtt{t} \mid (\mathtt{t}, M_1 : \mathbf{I} : M_2, e') \in \overline{K} \quad \delta'' = \mathsf{fold}(\mathsf{lift}\ \ell\ \mathtt{RL},\ \delta',\ \bar{\mathtt{t}'})}{\Delta, \delta, \overline{K} \mid\mid (\mathtt{t}, M, F[\mathtt{isolateEvt}(v)]) \Longrightarrow \Delta, \delta'', \overline{K} \mid\mid (\mathtt{t}, \mathbf{I} : M, F[v])}$$

SendRecv

$$\frac{\Delta' = \delta(\mathtt{t}_1) \diamond \delta(\mathtt{t}_2)}{\Delta, \delta, (\mathtt{t}_1, M_1, F_1[\mathtt{sendEvt}(c, v)]) \mid\mid (\mathtt{t}_2, M_2, F_2[\mathtt{recvEvt}(c)]) \mid\mid \overline{K} \Longrightarrow \Delta, \delta[\mathtt{t}_1 \mapsto \Delta', \mathtt{t}_2 \mapsto \Delta'], (\mathtt{t}_1, M_1, F_1[\mathtt{alwaysEvt\ unit}]) \mid\mid (\mathtt{t}_2, M_2, F_2[\mathtt{alwaysEvt}\ v]) \mid\mid \overline{K}}$$

NestedIsolateEvt

$$\frac{}{\Delta, \delta, \overline{K} \mid\mid (\mathtt{t}, M_1 : \mathbf{I} : M_2, F[\mathtt{isolateEvt}(v)]) \Longrightarrow \Delta, \delta, \overline{K} \mid\mid (\mathtt{t}, M_1 : \mathbf{I} : M_2, F[v])}$$

IsolateEvtComplete

$$\frac{}{\Delta, \delta, \overline{K} \mid\mid (\mathtt{t}, \mathbf{I} : M, \mathtt{alwaysEvt}\ v) \Longrightarrow \Delta, \delta, \overline{K} \mid\mid (\mathtt{t}, M, \mathtt{alwaysEvt}\ v)}$$

Fig. 13. Semantics for concurrent isolate evaluation using capabilities.

$||(\mathtt{t}_3, M_3, \mathcal{F}_3[\mathtt{sendEvt}(ch_2, v_2)])||(\mathtt{t}_4, M_4, \mathcal{F}_4[\mathtt{recvEvt}(ch_2)])||\overline{K}, \overline{T}\,\rangle \Longrightarrow$
$\langle\, \Delta, \delta, (\mathtt{t}_1, M_1, \mathcal{F}_1[\mathtt{sendEvt}(ch_1, v_1)])||(\mathtt{t}_2, M_2, \mathcal{F}_2[\mathtt{recvEvt}(ch_1)])$
$||(\mathtt{t}_3, M_3, \mathcal{F}_3[\mathtt{alwaysEvtunit}])||(\mathtt{t}_4, M_4, \mathcal{F}_4[\mathtt{alwaysEvt}v])||\overline{K}, \overline{T}\,\rangle \Longrightarrow$
$\langle\, \Delta, \delta, (\mathtt{t}_1, M_1, \mathcal{F}_1[\mathtt{alwaysEvt\ unit}])||(\mathtt{t}_2, M_2, \mathcal{F}_2[\mathtt{alwaysEvt}\ v])$
$||(\mathtt{t}_3, M_3, \mathcal{F}_3[\mathtt{alwaysEvtunit}])||(\mathtt{t}_4, M_4, \mathcal{F}_4[\mathtt{alwaysEvt}v])||\overline{K}, \overline{T}\,\rangle$

Lemma 6. *If*
$\langle\, \Delta, \delta, \overline{K}, \overline{T}\,\rangle \longmapsto \langle\, \Delta', \delta', \overline{K}', \overline{T}\,\rangle \longmapsto \langle\, \Delta', \delta', \overline{K}', \overline{T}'\,\rangle$
then
$\langle\, \Delta, \delta, \overline{K}, \overline{T}\,\rangle \longmapsto \langle\, \Delta, \delta, \overline{K}, \overline{T}'\,\rangle \longmapsto \langle\, \Delta, \delta, \overline{K}', \overline{T}'\,\rangle$

Lemma 7. *If*
$\langle\, \Delta, \delta, \overline{K}, \overline{T}\,\rangle \longmapsto \langle\, \phi, \phi, \phi, \overline{T}\ ||\ \overline{T}'\,\rangle \longmapsto \langle\, \phi, \phi, \phi, \overline{T}''\ ||\ \overline{T}'\,\rangle$
then
$\langle\, \Delta, \delta, \overline{K}, \overline{T}\,\rangle \longmapsto \langle\, \Delta, \delta, \overline{K}, \overline{T}''\,\rangle \longmapsto \langle\, \phi, \phi, \phi, \overline{T}''\ ||\ \overline{T}'\,\rangle$

Lemma 8. *If*
$\langle\, \Delta, \delta, \overline{K}, \overline{T} \,||\, (\mathtt{t}, E[\mathtt{sync}v])\,\rangle \longmapsto \langle\, \Delta, \delta, \overline{K} \,||\, (\mathtt{t}, E, v), \overline{T}\,\rangle \longmapsto \langle\, \Delta, \delta, \overline{K} \,||\, (\mathtt{t}, E, v), \overline{T}'\,\rangle$
then
$\langle\, \Delta, \delta, \overline{K}, \overline{T} \,||\, (\mathtt{t}, E[\mathtt{sync}v])\,\rangle \longmapsto \langle\, \Delta, \delta, \overline{K}, \overline{T}' \,||\, (\mathtt{t}, E[\mathtt{sync}v])\,\rangle \longmapsto \langle\, \Delta, \delta, \overline{K} \,||\, (\mathtt{t}, E, v), \overline{T}'\,\rangle$

Theorem 1. (**Soundness of Transactional Evaluation**)
If
$\Delta, \delta, \overline{K} \,||\, (\mathtt{t}, M, \mathcal{F}[\mathtt{isolateEvt}(v)]) \Longrightarrow \ldots \Longrightarrow \Delta', \delta', \overline{K}' \,||\, (\mathtt{t}, \mathbf{I} : M, \mathcal{F}[\mathtt{alwaysEvt}(v')])$
then
$\overline{K} \,||\, (\mathtt{t}, M, \mathcal{F}[\mathtt{isolateEvt}(v)])) \rightsquigarrow \ldots \rightsquigarrow \overline{K}' \,||\, (\mathtt{t}, \mathbf{I} : M, \mathcal{F}[\mathtt{alwaysEvt}(v')])$

Proof Theorem 1.
Base Case

Inductive Case

Theorem 2. (**Soundness**) *If*
$\langle\, \Delta, \delta, \overline{K} \,||\, (\mathtt{t}, M, \mathcal{F}[\mathtt{isolateEvt}(v)]), \overline{T}\,\rangle \longmapsto \ldots \longmapsto \langle\, \Delta', \delta', \overline{K}' \,||\, (\mathtt{t}, \mathbf{I} : M, \mathcal{F}[\mathtt{alwaysEvt}(v')]), \overline{T}'\,\rangle$
then
$\langle \overline{K} \,||\, (\mathtt{t}, M, \mathcal{F}[\mathtt{isolateEvt}(v)]), \overline{T}\rangle) \rightarrow \ldots \rightarrow \langle \overline{K}' \,||\, (\mathtt{t}, \mathbf{I} : M, \mathcal{F}[\mathtt{alwaysEvt}(v')]), \overline{T}'\rangle$
Proof Theorem 2.
Base Case

Assume:
$\langle\, \Delta, \delta, \overline{K} \,||\, (\mathtt{t}, M, \mathcal{F}[\mathtt{isolateEvt}(v)]), \overline{T}\,\rangle \longmapsto \langle\, \Delta', \delta', \overline{K}' \,||\, (\mathtt{t}, \mathbf{I} : M, \mathcal{F}[\mathtt{alwaysEvt}(v')]), \overline{T}\,\rangle$
By **Theorem 1.**

$\langle \overline{K} \,||\, (\mathtt{t}, M, \mathcal{F}[\mathtt{isolateEvt}(v)]), \overline{T}\rangle) \rightarrow \langle \overline{K}' \,||\, (\mathtt{t}, \mathbf{I} : M, \mathcal{F}[\mathtt{alwaysEvt}(v')]), \overline{T}\rangle$
□
Inductive Case

Assume **Theorem 2.** holds for evaluation sequences of length n:
We know:
$\langle\, \Delta, \delta, \overline{K} \;||\; (\mathtt{t}, M, \mathcal{F}[\mathtt{isolateEvt}(v)]), \overline{T}\,\rangle \longmapsto ... \longmapsto \langle\, \Delta'', \delta'', \overline{K}'', \overline{T}''\,\rangle$
If:
$\langle\, \Delta'', \delta'', \overline{K}'', \overline{T}''\,\rangle \longmapsto \langle\, \Delta', \delta', \overline{K}' \;||\; (\mathtt{t}, \mathbf{I} : M, \mathcal{F}[\mathtt{alwaysEvt}(v')]), \overline{T}'\,\rangle$
Need to show:
$\langle \overline{K}'', \overline{T}''\rangle \rightarrow \langle \overline{K}' \;||\; (\mathtt{t}, \mathbf{I} : M, \mathcal{F}[\mathtt{alwaysEvt}(v')]), \overline{T}'\rangle$
Case `Spawn`:
By definition of `Spawn`:
$\overline{T}'' = \overline{T}''' \;||\; (\mathtt{t}, E[\mathtt{spawn}\ e])$
$\overline{T}' = (\mathtt{t}', [e]) \;||\; (\mathtt{t}, E[\mathtt{unit}]) \;||\; \overline{T}'''$
$\overline{K}'' = \overline{K}' \;||\; (\mathtt{t}, \mathbf{I} : M, \mathcal{F}[\mathtt{alwaysEvt}(v')])$

Implies:
$\langle \overline{K}'', \overline{T}''\rangle \rightarrow \langle \overline{K}' \;||\; (\mathtt{t}, \mathbf{I} : M, \mathcal{F}[\mathtt{alwaysEvt}(v')]), \overline{T}'\rangle$
Case `SyncThread`:
By definition of `SyncThread`:
$e \hookrightarrow e'$
$\overline{T}'' = thread\mathtt{t}E[\mathtt{sync}\ v] \;||\; \overline{T}'$
$\overline{K}'' = \overline{K}''' \;||\; \;||\; (\mathtt{t}, \mathbf{I} : M, \mathcal{F}[\mathtt{alwaysEvt}(v')])$
$\overline{K}' = \overline{K}''' \;||\; (\mathtt{t}, E :\cdot, v)$
Implies:
$\langle \overline{K}'', \overline{T}''\rangle \rightarrow \langle \overline{K}' \;||\; (\mathtt{t}, \mathbf{I} : M, \mathcal{F}[\mathtt{alwaysEvt}(v')]), \overline{T}'\rangle$
Case `StepThread`:
By definition of `StepThread`:
$\overline{T}'' = (\mathtt{t}, E[e]) \;||\; \overline{T}'''$
$\overline{T}' = (\mathtt{t}, E[e']) \;||\; \overline{T}'''$
$\overline{K}'' = \overline{K}' \;||\; (\mathtt{t}, \mathbf{I} : M, \mathcal{F}[\mathtt{alwaysEvt}(v')])$

Implies:
$\langle \overline{K}'', \overline{T}''\rangle \rightarrow \langle \overline{K}' \;||\; (\mathtt{t}, \mathbf{I} : M, \mathcal{F}[\mathtt{alwaysEvt}(v')]), \overline{T}'\rangle$
Case `StepTransactionalThread`:
By definition of `StepTransactionalThread`:
$\Delta'', \delta'', \overline{K}'' \Longrightarrow Delta', \delta', \overline{K}' \;||\; (\mathtt{t}, \mathbf{I} : M, \mathcal{F}[\mathtt{alwaysEvt}(v')])$
By **Theorem 1.**:

$\overline{K}'' \rightarrow \overline{K}' \;||\; (\mathtt{t}, \mathbf{I} : M, \mathcal{F}[\mathtt{alwaysEvt}(v')])$
Case `CommitTransThread`:
By definition of `CommitTransThread` this reduction cannot be applied.
□

Bibliography

[1] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. In *PLDI*, pages 26–37, 2006.

[2] Joe Armstrong, Robert Virding, Claes Wikstrom, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 2nd edition, 1996.

[3] Colin Blundell, E. Christopher Lewis, and Milo Martin. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Computer Architecture Letters*, 5(2), 2006.

[4] The Objective Caml System Release 3.10, Event Module, 2007.

[5] Kevin Donnelly and Matthew Fluet. Transactional Events. *The Journal of Functional Programming*, pages 649–706, 2008.

[6] Laura Effinger-Dean, Matthew Kehrt, and Dan Grossman. Transactional Events for ML. In *ICFP*, pages 103–114, 2008.

[7] Matthew Flatt and Robert Bruse Findler. Kill-safe Synchronization Abstractions. In *PLDI*, pages 47–58, 2004.

[8] Glasgow Haskell Compiler. `http://www.haskell.org/ghc`.

[9] Tim Harris and Keir Fraser. Language support for Lightweight Transactions. In *OOPSLA*, pages 388–402, 2003.

[10] Tim Harris, Simon Marlow, Simon Peyton Jones, , and Maurice Herlihy. Composable Memory Transactions. In *PPoPP*, pages 48–60, 2005.

[11] Maurice Herlihy, Victor Luchangco, and Mark Moir. A Flexible Framework for Implementing Software Transactional Memory. In *OOPSLA*, pages 253–262, 2006.

[12] Maurice Herlihy, Victor Luchangco, Mark Moir, and III William Scherer. Software Transactional Memory for Dynamic-Sized Data Structures. In *PODC*, pages 92–101, 2003.

[13] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, pages 289–300, 1993.

[14] Aaron Kimball and Dan Grossman. Software Transactions Meet First-Class Continuations. In *Scheme Workshop*, pages 47–58, 2007.

[15] James Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2007.

[16] V.J. Marathe, William N. Scherer III, and Michael L. Scott. Design Tradeoffs in Modern Software Transactional Memory Systems. In *LCR*, pages 1–7, 2004.

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

[18] Michael F. Ringenburg and Dan Grossman. AtomCaml: First-class atomicity via Rollback. In *ICFP*, pages 92–104, 2005.

[19] George Russell. Events in Haskell, and How to Implement Them. In *ICFP*, pages 157–168, 2001.

[20] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a High-Performance Software Transactional Memory system for a Multi-Core Runtime. In *PPoPP*, pages 187–197, 2006.

[21] Nir Shavit and Dan Touitou. Software Transactional Memory. In *PODC*, pages 204–213, 1995.

[22] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing Isolation and Ordering in STM. In *PLDI*, pages 78–88, 2007.

[23] Yannis Smaragdakis, Anthony Kay, Reimer Behrends, and Michal Young. Transactions with Isolation and Cooperation. In *OOPSLA*, pages 191–210, 2007.

[24] Lukasz Ziarek, Philip Schatz, and Suresh Jagannathan. Stabilizers: A Modular Checkpointing Abstraction for Concurrent Functional Programs. In *ICFP*, pages 136–147, 2006.