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 (4; 5) 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) (14). 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, facilitating 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 (11) 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 (4; 5) 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) (14). 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 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. To the best of our knowledge, the description of isolates as given in this paper provides the first formal characterization of serializability for message-passing languages.

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 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, Section 8 presents related work, and Section 9 presents conclusions.
2 CML and Transactional Events

Concurrent ML (14) (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 hold: send(c, v) ≡ sync(sendEvt(c,v)) and recv(c) ≡ 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, chooseEvt[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 (4; 5). 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:

\[
\text{thenEvt(recvEvt(ch),}
\begin{array}{l}
\text{fn(v) => if g(v) then alwaysEvt(v)}
\text{else neverEvt)}
\end{array}
\]

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

While transactional events execute atomically, they do not enforce isolation, and thus cannot be used to provide the kind of transactional guarantees (i.e., serializability) found in shared-memory transactional abstractions. To illustrate the problem, consider a simple 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, augmented with transactional events, 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 that queries the server to check the number of resources it has, and based on the result, makes a request
datatype ops = Query of int chan | Res 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 => wrap (sendEvt(ch,n),
              fn () => n)
          | Res i => if n >= i
              then alwaysEvt(n-i)
              else neverEvt
          | Rel j => ...)
        in serverLoop(sync(evt))
      end
    in (spawn(fn () => serverLoop(n));
        reqCh)
  end

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

to reserve some number of them. The client uses transactional events to ensure
that the query and reservation execute atomically:

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

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 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
cannot commit only when the client’s transaction does. But, the client requires two
operations involving the server to be executed atomically, 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. 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 would 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. 3).

datatype ops = ... | Done

fun server (n) =
  let val reqCh = channel()
  fun evtLoop(n) =
    thenEvt(recvEvt(reqCh),
            fn req =>
                case req of
                Done => alwaysEvt(n)
                | _  => thenEvt (case req of
                                Query ch => ...
                                | Res i => ...
                                | Rel j => ...),
                      fn (n') => evtLoop(n')))

  fun serverLoop(n) =
    serverLoop(sync(evtLoop(n)))
in (spawn(fn () => serverLoop(n));
     reqCh)
  end

Fig. 3. A server implementation that accepts multiple requests as part of a given
transaction.

In this revised implementation, the inner server loop (called \texttt{evtLoop}) yields
a transactional event. The outer server loop (called \texttt{serverLoop}) executes a \texttt{sync}
operation on this event. The event built by \texttt{evtLoop} enables the server to receive
multiple requests defined as part of a client’s complex atomic action. The client
explicitly indicates completion of its transaction (via operation \texttt{Done}): when this
operation is received, the result of the transactional event (i.e., the new resource
state) is supplied as the argument for the next transaction.

Unfortunately, although this solution allows the server to accept multiple
requests issued as part of a client protocol, it does \textit{not} prevent the server from
accepting interleaved requests from different clients. Serializability is therefore
compromised. 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 perform-
ing 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. Even though the solution preserves atomicity – all of the clients’ actions will be performed or none of them will be – 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 and which ensures global progress, 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: $'a \text{evt} \rightarrow 'a \text{evt}$ 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:

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

assuming the server implementation given in Fig. 3. To prevent interference induced by such communication, isolate evaluation ensures that all communications to the server by $C_1$ are performed prior to all communication by $C_2$ or vice 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, and the threads they communicate with (like the server) are free to communicate with different isolates, 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. 4 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.

In our syntax (see Fig. 4) $v$ ranges over values, $c$ over channel references, $\gamma$ over constants, $e$ over expressions, and $t$ over thread identifiers. The semantics
shown in Fig. 5 defines three relations. The ↬ 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 →. A global state consists of a set of transactional threads (K) and non-transactional threads (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 (T). The StepThread rule simply allows local execution within a non-transactional thread.

A thread gets added to K when it attempts to synchronize an event (rule SyncThread). Thus, every event synchronization initiates a transaction. For our purposes here, complex events are only built using thenEvtS. A thread executing transactionally is represented as a triple, (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 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
<table>
<thead>
<tr>
<th>App</th>
<th>Channel</th>
<th>Spawn</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\lambda x.e)v \leftrightarrow e[v/x]$</td>
<td>$c$ fresh ( ch() \leftrightarrow c )</td>
<td>$t'$ fresh ( (K, (t, E[\text{spawn } e]) \parallel T) \rightarrow (K, (t', [c]) \parallel (t, E[\text{unit}]) \parallel T) )</td>
</tr>
</tbody>
</table>

### SyncThread

$$ \frac{K, (t, E[\text{sync } v]) \parallel T \rightarrow (t, E, v) \parallel K, T}{K \sim K'} \quad \frac{K \sim K'}{(K, T) \rightarrow (K', T)} $$

### StepThread

$$ \frac{e \leftrightarrow e'}{(K, (t, E[c]) \parallel T) \rightarrow (K, (t', E'[c']) \parallel T)} $$

### StepTransactionalThread

$$ \frac{K \sim K'}{(K, T) \rightarrow (K', T)} $$

### CommitTransThreads

$$ \frac{K = (t_1, E_1, \text{alwaysEvt } v_1) \parallel ... \parallel (t_n, E_n, \text{alwaysEvt } v_n)}{T' = T \parallel (t_1, E_1[v_1]) \parallel ... \parallel (t_n, E_n[v_n]) \parallel (K, T) \rightarrow (\phi, T')} $$

### StepRunThread

$$ \frac{e \leftrightarrow e'}{K \parallel (t, M, F[e]) \sim K \parallel (t, M, F[e'])} $$

### NestedSync

$$ \frac{K \parallel (t, M, F[\text{sync } v]) \sim K \parallel (t, F : M, v)}{(K, T) \rightarrow (K, F[v], T)} $$

### NestedSyncComplete

$$ \frac{K \parallel (t, F : M, \text{alwaysEvt } v) \sim K \parallel (t, M, F[v])}{(K, F[v], T) \rightarrow (K, T)} $$

### SendRecv

$$ \frac{(t_1, M_1, F_1[\text{sendEvt}(c, v)]) \parallel (t_2, M_2, F_2[\text{recvEvt}(c)]) \parallel K \sim (t_1, M_1, F_1[\text{alwaysEvt unit}]) \parallel (t_2, M_2, F_2[\text{alwaysEvt } v]) \parallel K}{(t_1, M_1, F_1[\text{sendEvt}(c, v)]) \parallel (t_2, M_2, F_2[\text{recvEvt}(c)]) \parallel K \sim (t_1, M_1, F_1[\text{alwaysEvt unit}]) \parallel (t_2, M_2, F_2[\text{alwaysEvt } v]) \parallel K} $$

### IsolateEvt

$$ \frac{(t_1, M_1 : I : M_2, e) \notin K}{K \parallel (t, M, F[\text{isolateEvt}(v)]) \sim K \parallel (t, I : M, F[v])} $$

### NestedIsolateEvt

$$ \frac{K \parallel (t, M, F[v]) \sim K \parallel (t, M, F[\text{isolateEvt}(v)])}{(K, T) \rightarrow (K, T')} $$

### IsolateEvtComplete

$$ \frac{K \parallel (t, M, \text{alwaysEvt } v) \sim K \parallel (t, M, \text{alwaysEvt } v)}{(K, T) \rightarrow (\phi, T')} $$

**Fig. 5.** A semantics for isolates and transactional events that prevents concurrent isolate evaluation.
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 \texttt{alwaysEvt}(v), the thread is guaranteed to be no longer
executing transactionally.

Transactional evaluation is defined via relation \( \leadsto \) 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 \textsc{StepRunThread} rule allows local reductions within transactions. Rule
\textsc{NestedSync} permits new transactions (initiated by a \texttt{sync} action) to be in-
stantiated within existing ones. The current context of the outer transaction is
recorded on the context transaction stack. When a transaction completes (rule
\textsc{NestedSyncComplete}), the saved context of the outer transaction is popped
from the stack and the value yielded by the completed inner transaction is sup-
plied 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 \texttt{alwaysEvt} value – an expression that yields \texttt{neverEvt} can never be
successfully synchronized.

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

Rule \textsc{IsolateEvt} defines isolate evaluation. An isolate combinator given an
event value enforces isolation on that event value by prohibiting concurrent eval-
uation of any other isolates; we record that a transactional thread is executing
an isolate by explicitly marking the context stack using token \texttt{I}. Rule \textsc{NestedIs-
olateEvt} allows a thread executing an isolate to initiate another one. When
an isolate completes, the thread state reverts to an ordinary transaction (rule
\textsc{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.
Unfortunately, simply removing this constraint to extract greater concurrency
can lead to incorrect (non-isolated) executions. Fig.6(a) depicts one such exa-
ple. Here, isolate \( \ell_a \) initiates communication with \( T_1 \) (and then \( T_2 \)); similarly,
 isolate \( \ell_b \) 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 \( \ell_a \) were sequenced before \( \ell_b \) (or \textit{vise versa})
the communication with thread \( T_2 \) (or conversely, \( T_1 \)) would be blocked.

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
Fig. 6. Serializability violations introduced by incorrect parallel evaluation of isolates.

view of the different isolates they communicate with. Consider the execution depicted in Fig. 6(b). Observe that $T_1$ witnesses an ordering in which $\ell_b$ executes before $\ell_a$. Similarly, $T_2$ observes a serialization in which $\ell_c$ occurs before $\ell_d$. Each of the isolates shown in the figure also communicate with $T_3$ and $T_4$. Thread $T_3$ participates in an action with $\ell_a$ before an action from $\ell_c$; $T_4$ participates in an action with $\ell_d$ before $\ell_b$; the communication between $T_3$ and $T_4$ (labeled edge 5) must enforce this ordering. There are now two classes serializable executions induced by these different actions. From $T_1$ and $T_4$’s perspective, we need to ensure an execution in which $\ell_b$ atomically executes before $\ell_a$ and $\ell_d$ atomically executes before $\ell_c$. On the other hand, from $T_2$ and $T_3$’s perspective, the required ordering is one in which $\ell_a$ atomically executes before $\ell_b$ and $\ell_c$ atomically executes before $\ell_d$.

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 also 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 $\ell$ denoting the dynamic instances of isolate event expressions, and tags. When an isolate event $\ell$ 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 ℓ, and the set \( R \) that abstracts the set of all other evaluating isolates. To ensure a communication action between a thread \( T \) and ℓ does not violate serializability invariants, 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 \( R \), 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 \( R \); allowing \( T \) to engage in a subsequent communication with ℓ would break obvious isolation guarantees on ℓ’s execution since ℓ could witness the effects of \( T \)’s interaction with other isolates.

Thus, suppose thread \( T \) has a capability \( ℓ \mapsto L \). 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 \( ℓ \mapsto R \), 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.

Examples Consider the two examples given in Fig. 7. In both diagrams, there is an isolate computation ℓ, and an abstract collection of isolate computations \( I \). Consider the execution shown in Fig. 7(a). ℓ performs two communication actions with thread \( T \); once these actions complete some isolate from the \( I \) initiates a communication as well. After the communication with ℓ, \( T \)’s capability map has the entry: \( ℓ \mapsto LR \). The entry indicates that \( T \) first engaged in a communication with ℓ and then with other isolates found in \( I \).

Fig. 7. Capabilities regulate communication with isolates.

Now, consider the more complex example shown in Fig. 7(b)). 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 $\ell$ results in $T$ acquiring a capability map containing capability $\ell \mapsto L$, indicating that $T$ has only communicated with $\ell$. The capability map is adjusted after the second communication via channel $y$. Now the capability map contains in addition the capability $\ell \mapsto LR$. When $\ell$ tries to communicate via $z$, a serializability violation occurs: the capability $\ell \mapsto LR$ indicates that $T$ has previously communicated with $\ell$ 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 $\ell$’s actions with respect to $T$ cannot be grouped in their entirety either before or after $I$’s.

5.2 Semantics

Our formulation is given in Figs. 8 and 9. The semantics makes use of a new relation $\triangleright$ (see Tag Enrichment) that captures the obvious ordering relationship among tags. Informally, tag $\iota$ can be enriched to tag $\iota'$ if $\iota$ imposes the same or fewer restrictions on isolate communication than $\iota'$. Note the presence of two additional tags $\hat{L}$ and $\hat{R}$ that have not been discussed thus far – these tags are used to build capabilities for threads executing isolate computations. If thread $T$ executes an isolate expression with label $\ell$, 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 $\ell$.

There are two other auxiliary definitions used in the semantics. The first, LIFTING, yields a new capability map that binds the isolate label $\ell$ to the supplied tag $\iota$. The second, BINDING, yields a new local capability map when new isolates are created (see rules ISOLATEFRESH and ISOLATECOMM in Fig. 9). Let $t$ be a thread that begins evaluation of a new isolate expression with label $\ell$. Its capability for the isolate is $\ell \mapsto \hat{L}$; $\delta'$ reflects the incorporation of this capability in $t'$ capability map. Moreover, all other evaluating threads that have previously communicated with other isolates (the set $\bar{t}'$) must have their capability map extended to have the capability $\ell \mapsto \hat{R}$; this capability indicates that all these threads do subsequently communicate with $\ell$, such communication must be serializable with respect to the isolates they have previously communicated with. Finally, threads currently executing isolate computations (the set $\bar{t}''$) must have their capability map augmented with the capability $\ell \mapsto \hat{R}$; this capability prevents these threads from communicating at all with $\ell$ since they are already executing an isolate computation. Note that we leverage the curried definition of lift in expressing the folds that specify the various capability maps.

Global evaluation is defined via relation $\rightarrow$ that is the analog of $\rightarrow$ in the semantics given in Fig. 5. Transactional evaluation specifically (defined by relation $\Rightarrow$) is now defined with respect to both global ($\Delta$) and local ($\delta$) 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.
Fig. 8. Capabilities and Capability Enrichment.

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 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 \( \iota \triangleright \iota' \) leverages the structure of the relation to allow a capability that is currently L or R to become 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. 9 presents the salient rules that define transactional evaluation; the appendix provides the full set. 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 \( K \). 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 la-
SyncThread

\[
\langle \Delta, \delta, K, (t, E_v^\text{sync}) \rangle \mapsto \langle \Delta, \delta, (t \mapsto \phi), (t, E \mapsto v) || K, T \rangle
\]

\[
\langle \Delta, \delta, T \rangle \mapsto \langle \phi, \phi, \phi, T' \rangle
\]

CommitTransThreads

\[
K = (t_1, E_1, \text{alwaysEvt } v_1) || \ldots || (t_n, E_n, \text{alwaysEvt } v_n)
\]

\[
T' = T || (t_1, E_1[v_1]) || \ldots || (t_n, E_n[v_n])
\]

IsolateFresh

\[
\ell \text{ fresh } \delta(t) = \phi
\]

\[
\text{bind } \delta, \ell, t, K = \delta'
\]

\[
\Delta, \delta, K \parallel (t, M, F[\text{isolateEvt}(v)]) \implies \Delta, \delta', K \parallel (t, I : M, F[v])
\]

IsolateComm

\[
\ell \text{ fresh } \delta(t) \neq \phi
\]

\[
\text{bind } \delta, \ell, t, K = \delta'
\]

\[
\Delta, \delta, K \parallel (t, M, F[\text{isolateEvt}(v)]) \implies \Delta[\ell \mapsto RL, \delta', K \parallel (t, I : M, F[v])
\]

SendRecv

\[
\delta(t_1) \diamond \delta(t_2)
\]

\[
\Delta, \delta, (t_1, M_1, F_1[\text{sendEvt}(c, v)]) \parallel (t_2, M_2, F_2[\text{recvEvt}(c)]) || K \implies \\
\Delta, \delta, (t_1, M_1, F_1[\text{alwaysEvt unit}]) \parallel (t_2, M_2, F_2[\text{alwaysEvt v}]) || K
\]

Fig. 9. Concurrent isolate evaluation using capabilities.

bel (\ell) is created and the \textit{bind} operation is used to set up appropriate tags as described above. 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, expressed as the condition \(\delta(t) \neq \phi\)). In this case, the newly created isolate should be serialized after any isolates 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 (RL) and stores the corresponding capability \(\ell \mapsto RL\) in the global map. Like IsolateFresh, the rule uses \textit{bind} to augment the capability map of other threads to include the appropriate capability for \(\ell\).

Isolation is enforced by the SendRecv rule shown below that prevents communication between threads that have either \(\hat{L}\) and \(\hat{R}\) 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. 8.

Rule SendRecv allows two threads to communicate provided their capability maps are \textit{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 \(\text{Dom}(I) = \text{Dom}(I')\) and \(I(\ell) \diamond I'(\ell)\) where \(L \diamond L, \hat{L} \diamond RL, \hat{R} \diamond R, \hat{R} \diamond LR\) and \(\iota \diamond \iota\); this rule is sym-
metric. For example, $L \L$ holds because a thread executing an isolate $\ell$ (thus having capability $\ell \rightarrow L$) can certainly communicate with a thread that has previously only communicated with this isolate (that thread would therefore have capability $\ell \rightarrow L$). As another example, if thread $T_1$ had previously communicated with isolate $\ell$ and thread $T_2$ had previously communicated with isolate $\ell'$, $T_1$ must acquire the capability for $\ell'$ maintained by $T_2$ and $T_2$ must acquire the capability for $\ell$ maintained by $T_1$. Capability enrichment allows this to happen, and thus prevents $T_1$ from subsequently communicating with $\ell'$ in ways inconsistent with tag associated with $T_2$'s capability on $\ell'$. 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.

**Examples** The first example shown in Fig. 10(a) is a refined version of Fig. 6(a). The isolate $\ell_a$ is created initially. Since there are no other isolates, it is mapped to $L$ in the capability map for the thread that executes it. Subsequently, isolate $\ell_b$ is created. The creation of the second isolate adds the capability $\ell_b \rightarrow R$ to $\ell_a$'s thread's capability map. Now, when thread $T$ communicates with isolates $\ell_a$, it inherits $\ell_a$'s bindings. No new bindings are introduced on the second communication with $\ell_a$, but when $T$ communicates to $\ell_b$, $T$'s capability for $\ell_b$ is lifted to $RL$.

The second example, shown in Fig. 10(b), illustrates capability enrichment in the case when two threads have communicated with two separate isolates. Like the previous example, after the creation of $\ell_b$, the thread executing $\ell_a$ has a local capability map with capabilities $\ell_a \rightarrow L$ and $\ell_b \rightarrow R$. When $\ell_a$ communicates with $T_1$, $T_1$ acquires capabilities $\ell_a \rightarrow L$ and $\ell_b \rightarrow R$. Similarly, when $T_2$ communicates with isolate $\ell_b$ its capability map becomes $\ell_b \rightarrow L$. 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 $\ell_a$ as $T_2$ can lift its capability for $\ell_a$ to $L$. However, for $\ell_b$, $T_1$ 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 $\ell_a$ – the serial ordering defined by the capability map dictates that communication with $\ell_a$ precede all communication with $\ell_b$.

### 6 Soundness

Our main soundness result (see Appendix C) asserts that concurrent isolate evaluation (as defined by Figs. 8 and 9) is equivalent to serial isolate evaluation (as defined by Fig. 5):
Fig. 10. Capabilities permit threads to communicate with multiple isolates, provided serializability invariants are preserved.

**Theorem.** If

\[ \Delta, \delta, \overline{K} \parallel (t, M, F[\text{isolateEvt}(v)]) \implies \ldots \implies \Delta', \delta', \overline{K}', \]  

where \[ \overline{K} = (t_1, M_1, F_1[\text{alwaysEvt}(v_1)]) \parallel \ldots \parallel (t_n, M_n, F_n[\text{alwaysEvt}(v_n)]) \]

then \[ \overline{K} \parallel (t, M, F[\text{isolateEvt}(v)]) \leadsto \ldots \leadsto \overline{K}' \]

The intuition underlying the proof for Theorem 6 is shown in Fig. 11; supporting lemmas are provided in the appendix. States \( S_1 \) through \( S_4 \) depict communication between an isolate with label \( \ell \) and a set of other isolates \( I \) with threads \( T_2, T_3 \) and \( T_4 \). In \( S_1 \), isolate \( \ell \) 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 \( \ell \). Even though the isolate \( \ell \) and the set of isolates \( I \) interleave their communication actions, their behavior is equivalent to an execution in which, for example, all the isolates in set \( I \) fully complete before \( \ell \). 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 \texttt{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. We briefly describe the structure of a prototype implemented in MLton (12), a whole program optimizing compiler for Standard ML that has been enriched to support isolated transactional events.
When a thread commences evaluation of an isolate executing within a transactional event, a new search thread is created (4; 5) 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 (20), 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 required capability binding, it simply inherits it from its communication 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

**Fig. 11.** Serializability of concurrent evaluation means all actions performed by an isolate can be permuted before the actions performed by another and vise versa.
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

Our work fully integrates a transactional semantics into the CML, a concurrent extension to SML that supports synchronous message passing (14). Previous work has leveraged the atomicity guarantee of transactional semantics to add expressivity to message-passing abstractions by introducing transactional events (4; 5). 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 (as isolates do), and therefore do not guarantee serializability of transactional computations.

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 (16), for Scheme in (6) and for Caml in (3). Asynchronous message-passing has been used in languages like Erlang (2). Transactional events (4; 5) have been implemented in Concurrent Haskell and CML, respectively.

There has been substantial work on formalizing software transactional memory for shared memory (1; 13; 9). Implementations for software transactional memory have been presented in functional languages such as Caml (15), Scheme (10) and Haskell (8). For example, the implementation of transactional events presented in (4) uses a software transactional memory extension of Concurrent Haskell (8).

There has also been other previous work in non-functional languages that has combined inter-thread communication and software transactions. In (19) the authors extend tradition transactional memory with the ability to observe the effects of other threads at selected points.

9 Conclusion

This paper explores a new event combinator that provides isolation of effects performed by concurrently executing transactional events. Although easy to describe, isolates add substantial expressive power, enabling the integration of a true transactional programming model into CML. The semantics also facilitates a reasonable implementation. We believe the addition of isolates to the CML program model is the first formal characterization of serializable software transactions for message-passing languages.
Bibliography

Appendix A: Complete Semantics for Concurrent Isolate Evaluation

\[\ell \in \text{Label} \quad \iota \in \text{Tag} := \cdot | \hat{L} | \hat{R} | L | R | LR | RL\]
\[\Delta \in \text{CapabilityMap} := \text{Label} \xrightarrow{\text{fin}} \text{Tag}\]
\[\delta \in \text{LocalCapabilityMap} := \text{TID} \xrightarrow{\text{fin}} \text{CapabilityMap}\]

\[I \diamond I' \text{ if } \text{Dom}(I) = \text{Dom}(I') \text{ and } I(\ell) \diamond I'(\ell) \text{ where } L \diamond L, L \diamond RL, R \diamond R, L \diamond R, R \diamond LR \text{ and } \iota \diamond \iota.\]

**Tag Enrichment**

\[\begin{array}{c}
\cdot \triangleright L \\
L \triangleright RL \\
R \triangleright LR
\end{array}\]

**Lifting**

\[
\text{lift } \ell \iota \delta t = \delta[t \mapsto \delta(t)[\ell \mapsto \iota]]
\]

**Binding**

\[\bar{t}' = \{ t | \delta(t) \neq \phi \} \quad \bar{t}'' = \{ t | (t, M_1 : I_1 : M_2, e') \in \bar{K} \}
\]
\[\delta' = \text{lift } \ell \delta \quad \delta'' = \text{fold}(\text{lift } \ell, \delta', \bar{t}')
\]

\[
\text{bind } \delta, \ell, t, \bar{K} = \text{fold}(\text{lift } \ell, \delta'', \bar{t}'')
\]

**Capability Enrichment**

\[\ell \notin \text{Dom}(\delta(t)) \quad \iota' \in \{L, R\} \quad \delta' = \text{lift } \ell \iota' \delta t
\]
\[\Delta, \delta, \bar{K} \parallel (t, M, e) \Rightarrow \Delta, \delta', \bar{K} \parallel (t, M, e)
\]

**Isolate Serialization**

\[\Delta(\ell) = \iota' \quad \delta t \ell = \iota \quad \iota \triangleright \iota' \quad \delta' = \text{lift } \ell \iota' \delta t
\]
\[\Delta, \delta, \bar{K} \parallel (t, M, e) \Rightarrow \Delta, \delta', \bar{K} \parallel (t, M, e)
\]

**Serialization Check**

\[
\Delta(\ell) = \iota' \quad \delta t \ell = \iota \quad \iota \triangleright \iota' \quad \delta' = \text{lift } \ell \iota' \delta t
\]
\[\Delta, \delta, \bar{K} \parallel (t, M, e) \Rightarrow \Delta, \delta', \bar{K} \parallel (t, M, e)
\]

Fig. 12. Capabilities and Capability Enrichment.
\( (\lambda x.e) \mapsto e[v/x] \)

\( \text{App} \)

\( c \_\text{fresh} \)

\( \text{Channel} \)

\( t' \_\text{fresh} \)

\( \text{Spawn} \)

\( (\Delta, \delta, K, (t, E[\text{unit}]) || T) \mapsto (\Delta, \delta, K, (t', e) || (t, E[\text{unit}]) || T) \)

\( \langle \Delta, \delta, K, (t, E[\text{sync } v]) || T \rangle \mapsto (\Delta, \delta, K, (t, E[\text{sync } v]) || K, T) \)

\( \langle \Delta, \delta, K \rangle \mapsto (\Delta', \delta', K') \)

\( \langle \Delta, \delta, K, T \rangle \mapsto (\Delta', \delta', K', T') \)

\( e \mapsto e' \)

\( \langle \Delta, \delta, K, (t, E[e]) || T \rangle \mapsto (\Delta, \delta, K, (t, E[e']) || T) \)

\( \text{CommitTransThreads} \)

\( K = (t_1, E_1, \text{alwaysEvt } v_1) || ... || (t_n, E_n, \text{alwaysEvt } v_n) \)

\( T' = T || (t_1, E_1[v_1]) || ... || (t_n, E_n[v_n]) \)

\( (\Delta, \delta, K, T) \mapsto (\phi, \phi, \phi, T') \)

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

\[
\begin{align*}
\ell & \leftrightarrow \ell' \\
\Delta, \delta, \overline{K} \parallel (t, M, F[e]) & \Rightarrow \Delta, \delta, \overline{K} \parallel (t, M, F[e'])
\end{align*}
\]

**NestedSync**

\[
\begin{align*}
\Delta, \delta, \overline{K} \parallel (t, M, F[\text{sync } v]) & \Rightarrow \\
\Delta, \delta, \overline{K} \parallel (t, F : M, v)
\end{align*}
\]

**NestedSyncComplete**

\[
\begin{align*}
\Delta, \delta, \overline{K} \parallel (t, M, \text{alwaysEvt } v) & \Rightarrow \\
\Delta, \delta, \overline{K} \parallel (t, M, F[v])
\end{align*}
\]

**ThenAlways**

\[
\begin{align*}
\Delta, \delta, \overline{K} \parallel (t, M, F[\text{thenEvt(alwaysEvt } (v_1), v_2)]) & \Rightarrow \\
\Delta, \delta, \overline{K} \parallel (t, M, F[v_2 \cdot v_1])
\end{align*}
\]

**SendRecv**

\[
\begin{align*}
\delta(t_1) \otimes \delta(t_2) \\
\Delta, \delta, (t_1, M_1, F_1[\text{sendEvt}(c, v)]) \parallel (t_2, M_2, F_2[\text{recvEvt}(c)]) \parallel \overline{K} & \Rightarrow \\
\Delta, \delta, (t_1, M_1, F_1[\text{alwaysEvt } \text{unit}]) \parallel (t_2, M_2, F_2[\text{alwaysEvt } v]) \parallel \overline{K}
\end{align*}
\]

**IsolateFresh**

\[
\begin{align*}
\ell & \text{ fresh } \delta(t) = \phi \\
\text{bind } \delta, \ell, t, \overline{K} & = \delta'
\end{align*}
\]

\[
\Delta, \delta, \overline{K} \parallel (t, M, F[\text{isolateEvt}(v)]) \Rightarrow \\
\Delta, \delta', \overline{K} \parallel (t, I : M, F[v])
\]

**IsolateComm**

\[
\begin{align*}
\ell & \text{ fresh } \delta(t) \neq \phi \\
\text{bind } \delta, \ell, t, \overline{K} & = \delta'
\end{align*}
\]

\[
\Delta, \delta, \overline{K} \parallel (t, M, F[\text{isolateEvt}(v)]) \Rightarrow \\
\Delta[\ell \mapsto \text{RL}], \delta', \overline{K} \parallel (t, I : M, F[v])
\]

**NestedIsolateEvt**

\[
\Delta, \delta, \overline{K} \parallel (t, M_1 : I : M_2, F[\text{isolateEvt}(v)]) \Rightarrow \\
\Delta, \delta, \overline{K} \parallel (t, M_1 : I : M_2, F[v])
\]

**IsolateEvtComplete**

\[
\Delta, \delta, \overline{K} \parallel (t, I : M, \text{alwaysEvt } v) \Rightarrow \\
\Delta, \delta, \overline{K} \parallel (t, M, \text{alwaysEvt } v)
\]

*Fig. 14.* Semantics for concurrent isolate evaluation using capabilities.
Appendix B. Supporting Lemmas

The proofs for the supporting lemmas and theorems, omitted here, but available in Appendix C follow from the explanations given below.

Consistency between the global capability map and the local capability maps is maintained at every step of evaluation.

Lemma 1. Given a state: \((\Delta, \delta, K, T)\)

\[
\forall \ell \in \text{dom}(\Delta) \text{ if } \Delta(\ell) = \text{LR} \text{ then } \delta_{t \ell} \neq \text{RL} \text{ Where } t \in K.
\]

Lemma 2. Given a state: \((\Delta, \delta, K, T)\)

\[
\forall \ell \in \text{dom}(\Delta) \text{ if } \Delta(\ell) = \text{RL} \text{ then } \delta_{t \ell} \neq \text{LR} \text{ Where } t \in K.
\]

The lemma holds by the definition of rules ISOLATE Serialization and Serialization Check.

\(P\) is a relation on concurrent isolate evaluation sequences. \(P(\Delta, \delta, K \Rightarrow \ldots \Rightarrow \Delta', \delta', K)\) is the set of evaluation sequences defined as the transitive closure over permutations of transition steps as defined by the lemmas. The lemmas define various single-step permutation actions on transactional threads.

Relation \(P\) is reflexive.

Lemma 3.

\[
\Delta, \delta, K \Rightarrow \ldots \Rightarrow \Delta', \delta', K' \in P(\Delta, \delta, K \Rightarrow \ldots \Rightarrow \Delta', \delta', K)
\]

Permuting the evaluation of expressions in two separate transactional threads via relation StepRunThread is permissible:

Lemma 4.

\[
\Delta, \delta, (t_1, M_1, e_1) || (t_2, M_2, e_2) || K \Rightarrow \Delta, \delta, (t_1, M_1, e'_1) || (t_2, M_2, e'_2) || K \Rightarrow \\
\Delta, \delta, (t_1, M_1, e_1) || (t_2, M_2, e_2) || K \Rightarrow \\
P(\Delta, \delta, (t_1, M_1, e_1) || (t_2, M_2, e_2) || K \Rightarrow \\
\Delta, \delta, (t_1, M_1, e'_1) || (t_2, M_2, e'_2) || K \Rightarrow \\
\Delta, \delta, (t_1, M_1, e'_1) || (t_2, M_2, e'_2) || K
\]

The lemma holds because the application of StepRunThread, which implies \(e \leftrightarrow e'\) via rule APP or CHANNEL, to a transactional thread is independent of all other threads in the program state. Thus in the initial state, StepRunThread can be applied to either \(t_1\) or \(t_2\). The application of the rule does not affect the other thread; therefore, the rule can be applied to the other thread in the resulting state. The final state is the same regardless of the order the rule was applied to the two transactional threads.

We are allowed to permute operations that extend the global capability map via ISOLATESerialization:
Lemma 5.
\( \Delta, \delta, K \upharpoonright (t_1, M_1, e_1) \upharpoonright (t_2, M_2, e_2) \Rightarrow \)
\( \Delta, \delta, K \upharpoonright (t_1, M_1, e_1) \upharpoonright (t_2, M_2, e_2) \Rightarrow \)
\( \Delta\ell \mapsto \ell', \delta''', K \upharpoonright (t_1, M_1, e_1) \upharpoonright (t_2, M_2, e_2) \Rightarrow \)
\( \Delta|\ell| \mapsto \ell', \ell''', K \upharpoonright (t_1, M_1, e_1) \upharpoonright (t_2, M_2, e_2) \in \)
\( P(\Delta, \delta, K \upharpoonright (t_1, M_1, e_1) \upharpoonright (t_2, M_2, e_2) \Rightarrow \)
\( \Delta|\ell| \mapsto \ell', \ell''', K \upharpoonright (t_1, M_1, e_1) \upharpoonright (t_2, M_2, e_2) \Rightarrow \)

The lemma holds because the two evaluation step via rule ISOLATESERIALIZATION are independent. The initial local capability map contains isolate labels \( \ell \) and \( \ell' \), such that \( \ell \not\in \Delta \) and \( \ell' \not\in \Delta \). Regardless of the order the two steps are applied, the final global capability map will contain both labels. Furthermore, the lifting for each label is independent of the other, and therefore the mappings will be the same as will the final local capability map (i.e. \( \delta''' \)).

Permuting a step via SendRecv and a step via StepRunThread, that evaluates another transactional thread (i.e. not one that was involved in the communication) is allowed:

Lemma 6.
\( \Delta, \delta, (t_1, M_1, e_1) \upharpoonright (t_2, M_2, F_2[sendEvt(ch, v)]) \upharpoonright (t_3, M_3, F_3[recvEvt(ch)]) \Rightarrow \)
\( \Delta, \delta, (t_1, M_1, e_1) \upharpoonright (t_2, M_2, F_2[sendEvt(ch, v)]) \upharpoonright (t_3, M_3, F_3[recvEvt(ch)]) \Rightarrow \)
\( \Delta, \delta, (t_1, M_1, e_1) \upharpoonright (t_2, M_2, F_2[alwaysEvt unit]) \upharpoonright (t_3, M_3, F_3[alwaysEvt v]) \Rightarrow \)
\( \Delta, \delta, (t_1, M_1, e_1) \upharpoonright (t_2, M_2, F_2[alwaysEvt unit]) \upharpoonright (t_3, M_3, F_3[alwaysEvt v]) \Rightarrow \)

The lemma holds because the two communicating threads evaluated in the SendRecv rule and the thread evaluated via rule StepRunThread are independent. Thus either rule can be applied to the initial state. If the SendRecv rule is applied first, it does not affect thread \( t_1 \) and if the StepRunThread is applied first, it does not affect communicating threads \( t_2 \) and \( t_3 \). Thus, the other rule can be applied to the resulting state. Regardless of which rule was applied first, the final state will be the same.

Permuting two SendRecv steps, each evaluating a different pair of communicating transactional threads, results in an equivalent program state:

Lemma 7.
\( \Delta, \delta, (t_1, M_1, F_1[sendEvt(ch_1, v_1)]) \upharpoonright (t_2, M_2, F_2[recvEvt(ch_1)]) \Rightarrow \)
\( \Delta, \delta, (t_1, M_1, F_1[sendEvt(ch_2, v_2)]) \upharpoonright (t_3, M_3, F_3[sendEvt(ch_2)]) \Rightarrow \)
\( \Delta, \delta, (t_1, M_1, F_1[alwaysEvt unit]) \upharpoonright (t_2, M_2, F_2[alwaysEvt v]) \Rightarrow \)
\( \Delta, \delta, (t_1, M_1, F_1[alwaysEvt unit]) \upharpoonright (t_3, M_3, F_3[alwaysEvt v]) \Rightarrow \)
\( \Delta, \delta, (t_1, M_1, F_1[sendEvt(ch_1, v_1)]) \upharpoonright (t_2, M_2, F_2[recvEvt(ch_1)]) \Rightarrow \)
\( \Delta, \delta, (t_1, M_1, F_1[sendEvt(ch_2, v_2)]) \upharpoonright (t_3, M_3, F_3[sendEvt(ch_2)]) \Rightarrow \)
\( \Delta, \delta, (t_1, M_1, F_1[alwaysEvt unit]) \upharpoonright (t_2, M_2, F_2[alwaysEvt v]) \Rightarrow \)
\( \Delta, \delta, (t_1, M_1, F_1[alwaysEvt unit]) \upharpoonright (t_3, M_3, F_3[alwaysEvt v]) \Rightarrow \)
The lemma holds because in the initial state either pair of communication actions can be evaluated via SendRecv, and neither one will effect the other pair of communicating actions. Therefore, regardless of the order the communications take place, the final state will be the same.

Transactional evaluation of an isolate in the concurrent isolate semantics as defined via relation $\Rightarrow$ is equivalent to the transactional evaluation of the same isolate in the serial isolate semantics as defined via relation $\sim$:

**Theorem 1. (Soundness of Transactional Evaluation)**

If

\[ \Delta, \delta, K \parallel (t, M, F[\text{isolateEvt}(v)]) \Rightarrow \ldots \Rightarrow \Delta'', \delta'', K'' \parallel (t, M, F[\text{alwaysEvt}(v)]) \]

then

\[ \exists \Delta, \delta, K \parallel (t, M, F[\text{isolateEvt}(v)]) \Rightarrow \ldots \Rightarrow \Delta', \delta', K' \Rightarrow \ldots \Rightarrow \]

\[ \Delta'', \delta'', K'' \parallel (t, M, F[\text{alwaysEvt}(v)]) \]

\[ \in P(\Delta, \delta, K) \parallel (t, M, F[\text{isolateEvt}(v)]) \Rightarrow \ldots \Rightarrow \Delta'', \delta'', K'' \parallel (t, M, F[\text{alwaysEvt}(v)]) \]

such that

\[ K' \parallel (t, M, F[\text{isolateEvt}(v)]) \sim \ldots \sim K'' \]

Intuitively, Theorem 1 holds because any concurrent evaluation of transactional threads performing an isolate can be permuted to an equivalent sequence such that no other isolate’s actions occur during the evaluation of the isolate. Such an evaluation sequence is equivalent to transactional evaluation of the isolate in the serial isolate semantics. The proof is an induction on transactional evaluation sequences from the initial state $\Delta, \delta, K$ to intermediate state $\Delta', \delta', K'$ and leverages the capability maps to ensure that such permutations can be performed while preserving program meaning.

**Theorem 2. (Soundness)**

If

\[ \Delta, \delta, K \parallel (t, M, F[\text{isolateEvt}(v)]) \Rightarrow \ldots \Rightarrow \Delta', \delta', K', \text{ where} \]

\[ K' = (t_1, M_1, F_1[\text{alwaysEvt}(v_1)]) \parallel \ldots \parallel (t_n, M_n, F_n[\text{alwaysEvt}(v_n)]) \]

then

\[ K' \parallel (t, M, F[\text{isolateEvt}(v)]) \sim \ldots \sim K'' \]

The proof of this theorem follows from Theorem 1 since $K'' \parallel (t, M, F[\text{alwaysEvt}(v)])$ from Theorem 1 can be $K'$ (from this theorem). We know by Theorem 1 that we can permute a concurrent evaluation sequence defined by $\Rightarrow$ such that there exists a serial evaluation sequence as defined by $\sim$. 
Appendix C. Proofs

Transactional evaluation of an isolate in the concurrent isolate semantics as defined via relation $\Rightarrow$ is equivalent to the transactional evaluation of the same isolate in the serial isolate semantics as defined via relation $\sim$:

Theorem 1. (Soundness of Transactional Evaluation)

If

$\Delta, \delta, K \parallel (t, M, F[isolateEvt(v)]) \Rightarrow \ldots \Rightarrow \Delta'', \delta'', K'' \parallel (t, M, F[alwaysEvt(v)])$

then

$\exists \Delta, \delta, K \parallel (t, M, F[isolateEvt(v)]) \Rightarrow \ldots \Rightarrow \Delta', \delta', K \Rightarrow \ldots \Rightarrow \Delta'', \delta'', K'' \parallel (t, M, F[alwaysEvt(v)]) \in P(\Delta, \delta, K \parallel (t, M, F[isolateEvt(v)]) \Rightarrow \ldots \Rightarrow \Delta'', \delta'', K'' \parallel (t, M, F[alwaysEvt(v)]))$

such that

$K \parallel (t, M, F[isolateEvt(v)]) \Rightarrow \ldots \Rightarrow K'$

Intuitively, Theorem 1 holds because any concurrent evaluation of transactional threads performing an isolate can be permuted to an equivalent sequence such that no other isolate’s actions occur during the evaluation of the isolate. Such an evaluation sequence is equivalent to transactional evaluation of the isolate in the serial isolate semantics. The proof is an induction on transactional evaluation sequences and leverages the lemmas presented above and the capability maps to ensure that such permutations can be performed while preserving program meaning.

Proof Theorem 1.

Base Case

WLOG we are given:

$\Delta, \delta, K \parallel (t, M, F[isolateEvt(v)]) \Rightarrow \Delta_1, \delta_1, K_1 \Rightarrow \ldots \Rightarrow \Delta'', \delta'', K'' \parallel (t, M, F[alwaysEvt(v)])$

Need to show:

$\Delta, \delta, K \parallel (t, M, F[isolateEvt(v)]) \Rightarrow \Delta', \delta', K' \Rightarrow \ldots \Rightarrow \Delta'', \delta'', K'' \parallel (t, M, F[alwaysEvt(v)])$

$\in P(\Delta, \delta, K \parallel (t, M, F[isolateEvt(v)]) \Rightarrow \Delta_1, \delta_1, K_1 \Rightarrow \ldots \Rightarrow \Delta'', \delta'', K'' \parallel (t, M, F[alwaysEvt(v)]))$

Such that:

$K \parallel (t, M, F[isolateEvt(v)]) \Rightarrow \ldots \Rightarrow K'$

Case CapabilityEnrichment

By definition of $P$:

$\Delta, \delta, K \parallel (t, M, F[isolateEvt(v)]) \Rightarrow \Delta_1, \delta_1, K_1 \Rightarrow \ldots \Rightarrow \Delta'', \delta'', K'' \parallel (t, M, F[alwaysEvt(v)])$

$\in P(\Delta, \delta, K \parallel (t, M, F[isolateEvt(v)]) \Rightarrow \Delta_1, \delta_1, K_1 \Rightarrow \ldots \Rightarrow \Delta'', \delta'', K'' \parallel (t, M, F[alwaysEvt(v)]))$

By definition of CapabilityEnrichment:

$K_1 = K \parallel (t, M, F[isolateEvt(v)])$
Case IsolateSerialization

By definition of $P$:

$$\Delta, \delta, \overline{K} \parallel (t, M, F[\text{isolateEvt}(v)]) \Rightarrow \Delta_1, \delta_1, \overline{K}_1 \Rightarrow \ldots \Rightarrow \Delta'', \delta'', \overline{K}'' \parallel (t, M, F[\text{alwaysEvt}(v)])$$

$$\in P(\Delta, \delta, \overline{K} \parallel (t, M, F[\text{isolateEvt}(v)]) \Rightarrow \Delta_1, \delta_1, \overline{K}_1 \Rightarrow \ldots \Rightarrow \Delta'', \delta'', \overline{K}'' \parallel (t, M, F[\text{alwaysEvt}(v)]))$$

By definition of IsolateSerialization:

$$\overline{K}_1 = \overline{K} \parallel (t, M, F[\text{isolateEvt}(v)])$$

Case SerializationCheck

By definition of $P$:

$$\Delta, \delta, \overline{K} \parallel (t, M, F[\text{isolateEvt}(v)]) \Rightarrow \Delta_1, \delta_1, \overline{K}_1 \Rightarrow \ldots \Rightarrow \Delta'', \delta'', \overline{K}'' \parallel (t, M, F[\text{alwaysEvt}(v)])$$

$$\in P(\Delta, \delta, \overline{K} \parallel (t, M, F[\text{isolateEvt}(v)]) \Rightarrow \Delta_1, \delta_1, \overline{K}_1 \Rightarrow \ldots \Rightarrow \Delta'', \delta'', \overline{K}'' \parallel (t, M, F[\text{alwaysEvt}(v)]))$$

By definition of SerializationCheck:

$$\overline{K}_1 = \overline{K} \parallel (t, M, F[\text{isolateEvt}(v)])$$

Case StepRunThread

By definition of $P$:

$$\Delta, \delta, \overline{K} \parallel (t, M, F[\text{isolateEvt}(v)]) \Rightarrow \Delta_1, \delta_1, \overline{K}_1 \Rightarrow \ldots \Rightarrow \Delta'', \delta'', \overline{K}'' \parallel (t, M, F[\text{alwaysEvt}(v)])$$

$$\in P(\Delta, \delta, \overline{K} \parallel (t, M, F[\text{isolateEvt}(v)]) \Rightarrow \Delta_1, \delta_1, \overline{K}_1 \Rightarrow \ldots \Rightarrow \Delta'', \delta'', \overline{K}'' \parallel (t, M, F[\text{alwaysEvt}(v)]))$$

SubCase App

$$\overline{K}_1 = \overline{K}'' \parallel (t, M, F[\text{isolateEvt}(v)]) \parallel (t', M', F'[e'])$$

By definition of App:

$$\Delta, \delta, \overline{K}'' \parallel (t, M, F[\text{isolateEvt}(v)]) \parallel (t', M', F'[e]) \Rightarrow \Delta, \delta, \overline{K}'' \parallel (t, M, F[\text{isolateEvt}(v)]) \parallel (t', M', F'[e'])$$

and $e = (\lambda x.e'')v$ and $e' = e''[v/x]$. Implies:

$$\overline{K}'' \parallel (t, M, F[\text{isolateEvt}(v)]) \parallel (t', M', F'[e']) \Rightarrow \overline{K}'' \parallel (t, M, F[\text{isolateEvt}(v)]) \parallel (t', M', F'[e']) \Rightarrow \overline{K}'' \parallel (t, M, F[\text{isolateEvt}(v)]) \parallel (t', M', F'[e']) \Rightarrow$$

SubCase Channel

$$\overline{K}_1 = \overline{K}'' \parallel (t, M, F[\text{isolateEvt}(v)]) \parallel (t', M', F'[e'])$$

By definition Channel:

$$\Delta, \delta, \overline{K}'' \parallel (t, M, F[\text{isolateEvt}(v)]) \parallel (t', M, F[e]) \Rightarrow \Delta, \delta, \overline{K}'' \parallel (t, M, F[\text{isolateEvt}(v)]) \parallel (t', M', F'[e'])$$

and $ch() \leftarrow c$ and $c = ch()$ and $e' = c$. Implies:

$$\overline{K}'' \parallel (t, M, F[\text{isolateEvt}(v)]) \parallel (t', M, F[e]) \Rightarrow$$
\[ 1 \quad K'' \mid \mid (t, M, F[\text{isolateEvt}(v)]) \mid \mid (t', M', F'[v']). \]

By definition of \( P \):
\[ \Delta, \delta, K \mid (t, M, F[\text{isolateEvt}(v)]) \Rightarrow \Delta_1, \delta_1, \overline{K}_1 \Rightarrow \ldots \Rightarrow \Delta'', \delta'', \overline{K}'' \mid (t, M, F[\text{alwaysEvt}(v)]) \]
\[ \in P(\Delta, \delta, K \mid (t, M, F[\text{isolateEvt}(v)]) \Rightarrow \Delta_1, \delta_1, \overline{K}_1 \Rightarrow \ldots \Rightarrow \Delta'', \delta'', \overline{K}'' \mid (t, M, F[\text{alwaysEvt}(v)])) \]

By definition NestedSync:
\[ K_1 = \overline{K}'' \mid (t, M, F[\text{isolateEvt}(v)]) \mid (t', M', F'[v]) \]

By definition NestedSyncComplete:
\[ \Delta, \delta, K' \mid (t, M, F[\text{isolateEvt}(v)]) \mid (t', M', F'[sync v]) \Rightarrow \Delta, \delta, K'' \mid (t, M, F[\text{isolateEvt}(v)]) \mid (t', F': M', v). \]

Implies:
\[ K'' \mid (t, M, F[\text{isolateEvt}(v)]) \mid (t', M', F'[sync v]) \sim \]
\[ K'' \mid (t, M, F[\text{isolateEvt}(v)]) \mid (t', F': M', v). \]

By definition of \( P \):
\[ \Delta, \delta, K \mid (t, M, F[\text{isolateEvt}(v)]) \Rightarrow \Delta_1, \delta_1, \overline{K}_1 \Rightarrow \ldots \Rightarrow \Delta'', \delta'', \overline{K}'' \mid (t, M, F[\text{alwaysEvt}(v)]) \]
\[ \in P(\Delta, \delta, K \mid (t, M, F[\text{isolateEvt}(v)]) \Rightarrow \Delta_1, \delta_1, \overline{K}_1 \Rightarrow \ldots \Rightarrow \Delta'', \delta'', \overline{K}'' \mid (t, M, F[\text{alwaysEvt}(v)])) \]

By definition NestedSyncComplete:
\[ K_1 = \overline{K}'' \mid (t, M, F[\text{isolateEvt}(v)]) \mid (t', M', F'[v]). \]

Implies:
\[ K'' \mid (t, M, F[\text{isolateEvt}(v)]) \mid (t', F': M', \text{alwaysEvt } v) \sim \]
\[ K'' \mid (t, M, F[\text{isolateEvt}(v)]) \mid (t', F'[v]). \]

By definition of \( P \):
\[ \Delta, \delta, K \mid (t, M, F[\text{isolateEvt}(v)]) \Rightarrow \Delta_1, \delta_1, \overline{K}_1 \Rightarrow \ldots \Rightarrow \Delta'', \delta'', \overline{K}'' \mid (t, M, F[\text{alwaysEvt}(v)]) \]
\[ \in P(\Delta, \delta, K \mid (t, M, F[\text{isolateEvt}(v)]) \Rightarrow \Delta_1, \delta_1, \overline{K}_1 \Rightarrow \ldots \Rightarrow \Delta'', \delta'', \overline{K}'' \mid (t, M, F[\text{alwaysEvt}(v)])) \]

By definition ThenAlways:
\[ K_1 = \overline{K}'' \mid (t, M, F[\text{isolateEvt}(v)]) \mid (t', M', F'[v_2 v_1]) \]

By definition NestedSyncComplete:
\[ \Delta, \delta, K' \mid (t, M, F[\text{isolateEvt}(v)]) \mid (t', M', F'[\text{thenEvt(alwaysEvt } v_1, v_2)]) \Rightarrow \Delta, \delta, K'' \mid (t, M, F[\text{isolateEvt}(v)]) \mid (t', M', F'[v_2 v_1]). \]

Implies:
\[ K'' \mid (t, M, F[\text{isolateEvt}(v)]) \mid (t', M', F'[\text{thenEvt(alwaysEvt } v_1, v_2)]) \sim \]
\[ K'' \mid (t, M, F[\text{isolateEvt}(v)]) \mid (t', M', F'[v_2 v_1]). \]

By definition of \( P \):
\[ \Delta, \delta, K \mid (t, M, F[\text{isolateEvt}(v)]) \Rightarrow \Delta_1, \delta_1, \overline{K}_1 \Rightarrow \ldots \Rightarrow \Delta'', \delta'', \overline{K}'' \mid (t, M, F[\text{alwaysEvt}(v)]) \]
\[ \forall (\Delta, \delta, s) \in (t, M, s) \] 

By definition (WLOG \( t \) may or may not equal \( t' \)):

\[ \Delta, \delta, s \Rightarrow (t', M', s) \Rightarrow \Delta', \delta', s' \Rightarrow (t', M', s') \]

By definition (WLOG \( t \) may or may not equal \( t' \)):

\[ \Delta, \delta, s \Rightarrow (t', M', s) \Rightarrow \Delta', \delta', s' \Rightarrow (t', M', s') \]

By definition SendRecv:

\[ \Delta, \delta, s \Rightarrow (t, M, s) \Rightarrow \Delta, \delta, s \Rightarrow (t, M, s) \]

By definition NestedIsolateEvt:

\[ \Delta, \delta, s \Rightarrow (t, M, s) \Rightarrow \Delta, \delta, s \Rightarrow (t, M, s) \]

By definition NestedIsolateEvt does not apply.

Case IsolateEvtComplete

By definition of SendComm rule does not apply.
\( K_1 = K'' \ || \ (t, M, F[\text{isolateEvt}(v)]) \ || \ (t', M', \text{alwaysEvt } v) \)

By definition of IsolateEvtComplete:
\( \Delta, \delta, K'' \ || \ (t, M, F[\text{isolateEvt}(v)]) \ || \ (t', I : M', \text{alwaysEvt } v) \implies \Delta, \delta, K''' \ || \ (t, M, F[\text{isolateEvt}(v)]) \ || \ (t', M', \text{alwaysEvt } v). \)

Implies:
\( K''' \ || \ (t, M, F[\text{isolateEvt}(v)]) \ || \ (t', I : M', \text{alwaysEvt } v) \sim K'' \ || \ (t, M, F[\text{isolateEvt}(v)]) \ || \ (t', M', \text{alwaysEvt } v). \)

**Inductive Case**

We know:
\[ \exists \Delta, \delta, K \ || \ (t, M, F[\text{isolateEvt}(v)]) \implies \ldots \]
\[ \Delta_n, \delta_n, K_n \implies \ldots \]
\[ \Delta'', \delta'', K'' \ || \ (t, M, F[\text{alwaysEvt } (v)]) \in \mathcal{P}(\Delta, \delta, K \ || \ (t, M, F[\text{isolateEvt}(v)]) \implies \ldots \Delta'', \delta'', K'' \ || \ (t, M, F[\text{alwaysEvt } (v)]) \]

Such that:
\[ K \ || \ (t, M, F[\text{isolateEvt}(v)]) \sim \ldots \sim K_n \]

Need to show:
\[ \exists \Delta, \delta, K \ || \ (t, M, F[\text{isolateEvt}(v)]) \implies \ldots \]
\[ \Delta_n, \delta_n, K_n \implies \Delta_{n+1}, \delta_{n+1}, K_{n+1} \implies \ldots \implies \Delta'', \delta'', K'' \ || \ (t, M, F[\text{alwaysEvt } (v)]) \in \mathcal{P}(\Delta, \delta, K \ || \ (t, M, F[\text{isolateEvt}(v)]) \implies \ldots \Delta'', \delta'', K'' \ || \ (t, M, F[\text{alwaysEvt } (v)]) \]

Such that:
\[ K \ || \ (t, M, F[\text{isolateEvt}(v)]) \sim \ldots \sim K_n \sim K_{n+1} \]

**Case CapabilityEnrichment**

By Induction:
\[ \exists \Delta, \delta, K \ || \ (t, M, F[\text{isolateEvt}(v)]) \implies \ldots \implies \Delta_n, \delta_n, K_n \implies \Delta_{n+1}, \delta_{n+1}, K_{n+1} \implies \ldots \implies \Delta'', \delta'', K'' \ || \ (t, M, F[\text{alwaysEvt } (v)]) \in \mathcal{P}(\Delta, \delta, K \ || \ (t, M, F[\text{isolateEvt}(v)]) \implies \ldots \Delta'', \delta'', K'' \ || \ (t, M, F[\text{alwaysEvt } (v)]) \]

By definition of CapabilityEnrichment:
\[ K_n = K_{n+1} \]

**Case IsolateSerialization**

By Induction:
\[ \exists \Delta, \delta, K \ || \ (t, M, F[\text{isolateEvt}(v)]) \implies \ldots \implies \Delta_n, \delta_n, K_n \implies \Delta_{n+1}, \delta_{n+1}, K_{n+1} \implies \ldots \implies \Delta'', \delta'', K'' \ || \ (t, M, F[\text{alwaysEvt } (v)]) \in \mathcal{P}(\Delta, \delta, K \ || \ (t, M, F[\text{isolateEvt}(v)]) \implies \ldots \Delta'', \delta'', K'' \ || \ (t, M, F[\text{alwaysEvt } (v)]) \]

By definition of IsolateSerialization:
\[ K_n = K_{n+1} \]

**Case SerializationCheck**
By Induction:
\[ \exists \Delta, \delta, K \mid (t, M, F[\text{isolEvt}(v)]) \Rightarrow \ldots \Rightarrow \]
\[ \Delta_n, \delta_n, K_n \Rightarrow \Delta_{n+1}, \delta_{n+1}, K_{n+1} \Rightarrow \ldots \Rightarrow \]
\[ \Delta'', \delta'', K'' \mid (t, M, F[\text{alwaysEvt}(v)]) \in \mathcal{P}(\Delta, \delta, K) \mid (t, M, F[\text{isolEvt}(v)]) \Rightarrow \ldots \Rightarrow \Delta'', \delta'', K'' \mid (t, M, F[\text{alwaysEvt}(v)]) \]

By definition of SerializationCheck:
\[ K_n = K'_{n+1} \]

Case StepRunThread

By Induction:
\[ \exists \Delta, \delta, K \mid (t, M, F[\text{isolEvt}(v)]) \Rightarrow \ldots \Rightarrow \]
\[ \Delta_n, \delta_n, K_n \Rightarrow \Delta_{n+1}, \delta_{n+1}, K_{n+1} \Rightarrow \ldots \Rightarrow \]
\[ \Delta'', \delta'', K'' \mid (t, M, F[\text{alwaysEvt}(v)]) \in \mathcal{P}(\Delta, \delta, K) \mid (t, M, F[\text{isolEvt}(v)]) \Rightarrow \ldots \Rightarrow \Delta'', \delta'', K'' \mid (t, M, F[\text{alwaysEvt}(v)]) \]

SubCase App

By definition of App:
\[ K'_n = K \mid (t, M, F[e]) \]
\[ K'_{n+1} = K \mid (t, M, F[e']) \]
\[ \Delta, \delta, K \mid (t, M, F[e]) \Rightarrow \]
\[ \Delta, \delta, K \mid (t, M, F[e']) \]
\[ \Delta, \delta, K \mid (t, M, F[e']) \] and \((\lambda x.e''[v/x] = e''[v] \text{ and } e = (\lambda x.e''[v])x \text{ and } e' = e''[v/x])\)

Implies:
\[ K \mid (t, M, F[e]) \Rightarrow K \mid (t, M, F[e']) \]

SubCase Channel

By definition of Channel:
\[ K'_n = K \mid (t, M, F[e]) \]
\[ K'_{n+1} = K \mid (t, M, F[e']) \]
\[ \Delta, \delta, K \mid (t, M, F[e]) \Rightarrow \]
\[ \Delta, \delta, K \mid (t, M, F[e']) \]
\[ \Delta, \delta, K \mid (t, M, F[e']) \] and \(\text{ch}() \leftarrow c \text{ and } e = \text{ch}() \text{ and } e = c\)

Implies:
\[ K \mid (t, M, F[e]) \Rightarrow K \mid (t, M, F[e']) \]

Case NestedSync

By Induction:
\[ \exists \Delta, \delta, K \mid (t, M, F[\text{isolEvt}(v)]) \Rightarrow \ldots \Rightarrow \]
\[ \Delta_n, \delta_n, K_n \Rightarrow \Delta_{n+1}, \delta_{n+1}, K_{n+1} \Rightarrow \ldots \Rightarrow \]
\[ \Delta'', \delta'', K'' \mid (t, M, F[\text{alwaysEvt}(v)]) \in \mathcal{P}(\Delta, \delta, K) \mid (t, M, F[\text{isolEvt}(v)]) \Rightarrow \ldots \Rightarrow \Delta'', \delta'', K'' \mid (t, M, F[\text{alwaysEvt}(v)]) \]

By definition of NestedSync:
\[ K_n = K \mid (t, M, F[\text{sync v}]) \]
\[ K'_{n+1} = K \mid (t, F : M, v) \]
\[ \Delta, \delta, K \mid (t, M, F[\text{sync v}]) \Rightarrow \Delta, \delta, K \mid (t, F : M, v) \]

Implies:
\[ K \mid (t, M, F[\text{sync v}]) \Rightarrow K \mid (t, F : M, v) \]

Case NestedSyncComplete
By Induction:
\[ \exists \Delta, \delta, \kappa || (t, M, F[\text{isolateEvt}(v)]) \implies \ldots \implies \]
\[ \Delta_n, \delta_n, \kappa_n \implies \Delta_{n+1}, \delta_{n+1}, \kappa'_{n+1} \implies \ldots \implies \]
\[ \Delta'', \delta'', \kappa'' || (t, M, F[\text{alwaysEvt}(v)]) \in \]
\[ P(\Delta, \delta, \kappa || (t, M, F[\text{isolateEvt}(v)]) \implies \ldots \implies \Delta'', \delta'', \kappa'' || (t, M, F[\text{alwaysEvt}(v)])) \]

By definition of NestedSyncComplete:
\[ \kappa_n = \kappa || (t, F : M, \text{alwaysEvt} v) \]
\[ \kappa_{n+1} = \kappa || (t, M, F[v]) \]
\[ \Delta, \delta, \kappa || (t, F : M, \text{alwaysEvt} v) \implies \Delta, \delta, \kappa || (t, M, F[v]) \]
Implies:
\[ \kappa || (t, F : M, \text{alwaysEvt} v) \sim \kappa || (t, M, F[v]) \]
Case ThenAlways
By Induction:
\[ \exists \Delta, \delta, \kappa || (t, M, F[\text{isolateEvt}(v)]) \implies \ldots \implies \]
\[ \Delta_n, \delta_n, \kappa_n \implies \Delta_{n+1}, \delta_{n+1}, \kappa'_{n+1} \implies \ldots \implies \]
\[ \Delta'', \delta'', \kappa'' || (t, M, F[\text{alwaysEvt}(v)]) \in \]
\[ P(\Delta, \delta, \kappa || (t, M, F[\text{isolateEvt}(v)]) \implies \ldots \implies \Delta'', \delta'', \kappa'' || (t, M, F[\text{alwaysEvt}(v)])) \]

By definition of Case ThenAlways:
\[ \kappa_n = \kappa || (t, M, F[\text{thenEvt}(\text{alwaysEvt}(v_1), v_2)]) \]
\[ \kappa_{n+1} = \kappa || (t, M, F[v_2 v_1]) \]
\[ \Delta, \delta, \kappa || (t, M, F[\text{thenEvt}(\text{alwaysEvt}(v_1), v_2)]) \implies \Delta, \delta, \kappa || (t, M, F[v_2 v_1]) \]
Implies:
\[ \kappa || (t, M, F[\text{thenEvt}(\text{alwaysEvt}(v_1), v_2)]) \sim \kappa || (t, M, F[v_2 v_1]) \]
Case IsolateFresh
By Induction:
\[ \exists \Delta, \delta, \kappa || (t, M, F[\text{isolateEvt}(v)]) \implies \ldots \implies \]
\[ \Delta_n, \delta_n, \kappa_n \implies \Delta_{n+1}, \delta_{n+1}, \kappa'_{n+1} \implies \ldots \implies \]
\[ \Delta'', \delta'', \kappa'' || (t, M, F[\text{alwaysEvt}(v)]) \in \]
\[ P(\Delta, \delta, \kappa || (t, M, F[\text{isolateEvt}(v)]) \implies \ldots \implies \Delta'', \delta'', \kappa'' || (t, M, F[\text{alwaysEvt}(v)])) \]

If \((t_1, M_1 : I : M_2, e) \notin \kappa_n \) and by definition of IsolateFresh:
\[ \kappa_n = \kappa || (t, M, F[\text{isolateEvt}(v)]) \]
\[ \kappa_{n+1} = \kappa || (t, I : M, F[v]) \]
\[ \Delta, \delta, \kappa || (t, M, F[\text{isolateEvt}(v)]) \implies \]
\[ \Delta, \delta', \kappa || (t, I : M, F[v]) \]
Implies:
\[ \kappa_n = \kappa || (t, M, F[\text{isolateEvt}(v)]) \]
\[ \kappa_{n+1} = \kappa || (t, I : M, F[v]) \]
\[ \kappa || (t, I : M, F[v]) \sim \]
\[ \kappa || (t, I : M, F[v]) \]

Else \((t_1, M_1 : I : M_2, e) \in \kappa_n \):

Let \(\ell\) be the isolate tag generated by IsolateFresh.
Case \(\ell \notin \text{dom}(\Delta')\):
We know by Lemma 1 and 2:
\[ \forall t \in K'' : \delta t \ell = R | L | \bar{R} | \bar{L} \]

By definition of \( P \) need to examine communications as only communications prevent permutation:
By definition of \( \text{SendRecv} \) two threads \( t_1 \) and \( t_2 \) can communicate if \( \delta(t_1) \circ \delta(t_1) \).
By definition of \( \circ, R, R \) and \( \bar{R} \) cannot communicate with \( L, L \) and \( L \).
By definition of \( P \) all actions of isolate \( \ell \) can be permuted.

Implies:
\[ \exists \Delta, \delta, K' || (t, M, F[\text{isolateEvt}(v)]) \implies ... \implies \]
\[ \Delta_n, \delta_n, K_n' \implies \Delta_n+1, \delta_n+1, K_n+1 \implies ... \implies \]
\[ P(\Delta, \delta, K' || (t, M, F[\text{alwaysEvt}(v)])) \in \]
\[ P(\Delta', \delta', K' || (t, M, F[\text{isolateEvt}(v)])) \implies ... \implies \Delta', \delta', K' || (t, M, F[\text{alwaysEvt}(v)]) \]

Such that:
\[ K' || (t, M, F[\text{isolateEvt}(v)]) \sim ... \sim K_n \sim K'_{n+1} \]

Case \( \Delta''(\ell) = n \):
We know by Lemma 1 and 2:
\[ \forall t \in K'' : \delta t \ell = R | L | \bar{R} | \bar{L} | \ell \]

By definition of \( P \) need to examine communications as only communications prevent permutation:
By definition of \( \text{SendRecv} \) two threads \( t_1 \) and \( t_2 \) can communicate if \( \delta(t_1) \circ \delta(t_1) \).
By definition of \( \circ, R, R \) and \( \bar{R} \) cannot communicate with \( L, L \) and \( R \).
By definition of \( P \) all actions of isolate \( \ell \) can be permuted completely prior or after all other isolates in \( K_n \).

Implies:
\[ \exists \Delta, \delta, K' || (t, M, F[\text{isolateEvt}(v)]) \implies ... \implies \]
\[ \Delta_n, \delta_n, K_n' \implies \Delta_n+1, \delta_n+1, K_n+1 \implies ... \implies \]
\[ P(\Delta, \delta, K' || (t, M, F[\text{alwaysEvt}(v)])) \in \]
\[ P(\Delta', \delta', K' || (t, M, F[\text{isolateEvt}(v)])) \implies ... \implies \Delta', \delta', K' || (t, M, F[\text{alwaysEvt}(v)]) \]

Such that:
\[ K' || (t, M, F[\text{isolateEvt}(v)]) \sim ... \sim K_n \sim K'_{n+1} \]

Case IsolateComm
By Induction:
\[ \exists \Delta, \delta, K' || (t, M, F[\text{isolateEvt}(v)]) \implies ... \implies \]
\[ \Delta_n, \delta_n, K_n' \implies \Delta_n+1, \delta_n+1, K_n+1 \implies ... \implies \]
\[ P(\Delta, \delta, K' || (t, M, F[\text{alwaysEvt}(v)])) \in \]
\[ P(\Delta', \delta', K' || (t, M, F[\text{isolateEvt}(v)])) \implies ... \implies \Delta', \delta', K' || (t, M, F[\text{alwaysEvt}(v)]) \]

By definition of IsolateComm: \( \delta(t) \neq \phi \)
Implies: \((t_1, M_1 : I : M_2, e) \in K_n \)
Let \( \ell \) be the isolate tag generated by IsolateComm.
\[ \Delta''(\ell) = RL \]

We know by Lemma 1 and 2:
\[ \forall t \in K'' \quad \delta \quad \ell \quad = \quad R \mid L \mid \hat{R} \mid \hat{L} \mid LR \]

By definition of \( \mathcal{P} \) need to examine communications as only communications prevent permutation:
By definition of \( \text{SendRecv} \) two threads \( t_1 \) and \( t_2 \) can communicate if \( \delta(t_1) \circ \delta(t_1) \).
By definition of \( \circ, \hat{R}, R, \) and \( LR \) cannot communicate with \( \hat{L} \) and \( L \).
By definition of \( \mathcal{P} \) all actions of isolate \( \ell \) can be permuted completely after all other isolates in \( K_n \).

Implies:
\[ \exists \Delta, \delta, K \mid (t, M, F[isolateEvt(v)]) \Rightarrow \ldots \Rightarrow \Delta_n, \delta_n, K_n \Rightarrow \Delta_{n+1}, \delta_{n+1}, K_{n+1} \Rightarrow \ldots \Rightarrow \]
\[ \Delta'', \delta'', K'' \mid (t, M, F[alwaysEvt (v)]) \in \mathcal{P}(\Delta, \delta, K) \mid (t, M, F[isolateEvt(v)]) \Rightarrow \ldots \Rightarrow \Delta'', \delta'', K'' \mid (t, M, F[alwaysEvt (v)]) \]

Such that:
\[ K \mid (t, M, F[isolateEvt(v)]) \sim \ldots \sim K_n \sim K_{n+1} \]
Case SendRecv

By Induction:
\[ \exists \Delta, \delta, K \mid (t, M, F[isolateEvt(v)]) \Rightarrow \ldots \Rightarrow \Delta_n, \delta_n, K_n \Rightarrow \Delta_{n+1}, \delta_{n+1}, K_{n+1} \Rightarrow \ldots \Rightarrow \]
\[ \Delta'', \delta'', K'' \mid (t, M, F[alwaysEvt (v)]) \in \mathcal{P}(\Delta, \delta, K) \mid (t, M, F[isolateEvt(v)]) \Rightarrow \ldots \Rightarrow \Delta'', \delta'', K'' \mid (t, M, F[alwaysEvt (v)]) \]

By definition:
\[ K_n = (t_1, M_1, F_1[sendEvt(c, v)]) \mid (t_2, M_2, F_2[recvEvt(c)]) \mid K \]
\[ K_{n+1} = (t_1, M_1, F_1[alwaysEvt unit]) \mid (t_2, M_2, F_2[alwaysEvt v]) \mid K \]
\[ \Delta, \delta, (t_1, M_1, F_1[sendEvt(c, v)]) \mid (t_2, M_2, F_2[recvEvt(c)]) \mid K \]
\[ \Delta, \delta, (t_1, M_1, F_1[alwaysEvt unit]) \mid (t_2, M_2, F_2[alwaysEvt v]) \mid K \]
Implies:
\[ (t_1, M_1, F_1[sendEvt(c, v)]) \mid (t_2, M_2, F_2[recvEvt(c)]) \mid K \sim \]
\[ (t_1, M_1, F_1[alwaysEvt unit]) \mid (t_2, M_2, F_2[alwaysEvt v]) \mid K \]
Case NestedIsolateEvent

By Induction:
\[ \exists \Delta, \delta, K \mid (t, M, F[isolateEvt(v)]) \Rightarrow \ldots \Rightarrow \Delta_n, \delta_n, K_n \Rightarrow \Delta_{n+1}, \delta_{n+1}, K_{n+1} \Rightarrow \ldots \Rightarrow \]
\[ \Delta'', \delta'', K'' \mid (t, M, F[alwaysEvt (v)]) \in \mathcal{P}(\Delta, \delta, K) \mid (t, M, F[isolateEvt(v)]) \Rightarrow \ldots \Rightarrow \Delta'', \delta'', K'' \mid (t, M, F[alwaysEvt (v)]) \]

By definition of NestedIsolateEvent:
\[ K_n = K \mid (t_1, M_1 : I : M_2, F[isolateEvt(v)]) \]
\[ K_{n+1} = K \mid (t_1, M_1 : I : M_2, F[v]) \]
\[ \Delta, \delta, K \mid (t_1, M_1 : I : M_2, F[isolateEvt(v)]) \Rightarrow \]
\[ \Delta, \delta, K \mid (t, M_1 : I : M_2, F[v]) \]
Implies:
\[ K \parallel (t, M_1 : I : M_2, F[\text{isolateEvt}(v)]) \sim K \parallel (t, M_1 : I : M_2, F[v]) \]

Case \textbf{IsolateEvtComplete}

By Induction:
\[ \exists \Delta, \delta, K \parallel (t, M, F[\text{isolateEvt}(v)]) \implies \ldots \implies \]
\[ \Delta_n, \delta_n, K \implies \Delta_{n+1}, \delta_{n+1}, K_{n+1} \implies \ldots \implies \]
\[ \Delta'', \delta'', K'' \parallel (t, M, F[\text{alwaysEvt}(v)]) \in \]
\[ P(\Delta, \delta, K \parallel (t, M, F[\text{isolateEvt}(v)]) \implies \ldots \implies \Delta'', \delta'', K'' \parallel (t, M, F[\text{alwaysEvt}(v)]) \]

By definition of \textbf{IsolateEvtComplete}:
\[ K_n = K \parallel (t, I : M, \text{alwaysEvt}(v)) \]
\[ K_{n+1} = K \parallel (t, M, \text{alwaysEvt}(v)) \]
\[ \Delta, \delta, K \parallel (t, I : M, \text{alwaysEvt}(v)) \implies \]
\[ \Delta, \delta, K \parallel (t, M, \text{alwaysEvt}(v)) \]

Implies:
\[ K \parallel (t, I : M, \text{alwaysEvt}(v)) \sim K \parallel (t, M, \text{alwaysEvt}(v)) \]
\[ \Box \]