Software Transactional Memory with Interactions*

Marino Miculan[1] and Marco Peressotti[2]

[1] University of Udine, Italy marino.miculan@uniud.it
[2] University of Southern Denmark, Denmark peressotti@imada.sdu.dk

Abstract Software Transactional memory (STM) is an emerging abstraction for concurrent programming alternative to lock-based synchronizations. Most STM models admit only *isolated* transactions, which are not adequate in multithreaded programming where transactions need to interact via shared data *before* committing. To overcome this limitation, in this paper we present *Atomic Transactional Memory* (ATM), a programming abstraction supporting *safe, data-driven interactions* between *composable* memory transactions. This is achieved by relaxing isolation between transactions, still ensuring atomicity. This model allows for *loosely-coupled* interactions since transaction merging is driven only by accesses to shared data, with no need to specify participants beforehand.

1 Introduction

Modern multicore architectures have emphasized the importance of abstractions supporting correct and scalable multi-threaded programming. In this model, threads can collaborate by interacting on shared data structures, such as tables, message queues, buffers, etc., whose access must be regulated to avoid inconsistencies. Traditional lock-based mechanisms (like semaphores and monitors) are notoriously difficult and error-prone, as they easily lead to deadlocks, race conditions and priority inversions; moreover, they are not composable and hinder parallelism, thus reducing efficiency and scalability. *Transactional memory* (TM) has emerged as a promising abstraction to replace locks [5, 19]. The basic idea is to mark blocks of code as *atomic*; then, execution of each block will appear either as if it was executed sequentially and instantaneously at some unique point in time, or, if aborted, as if it did not execute at all. This is obtained by means of *optimistic* executions: these blocks are allowed to run concurrently, and eventually if an interference is detected a block is automatically restarted after that its effects are rolled back. Thus, each transaction can be viewed in isolation as a *single-threaded* computation, significantly reducing the programmer's burden.

[*] Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0). Supported by Italian MIUR project PRIN 2017FTXR7S *IT MATTERS* (Methods and Tools for Trustworthy Smart Systems) (M. Miculan), and by the Independent Research Fund Denmark, Natural Sciences, grant DFF-7014-00041 (M. Peressotti).

Moreover, transactions are composable and ensure absence of deadlocks and priority inversions, automatic roll-back on exceptions, and increased concurrency.

However, in multi-threaded programming transactions may need to interact and exchange data *before* committing. In this situation, transaction isolation is a severe shortcoming. A simple example is a request-response interaction between two transactions via a shared buffer. We could try to synchronize the threads accessing the buffer `b` by means of two semaphores `c1`, `c2` as follows:

```
// Party1 (Master)                        // Party2 (Worker)
atomically {                              atomically {
  <put request in b>                        down(c1); // wait for data
  up(c1);                                   <get request from b>
  <some other code; may abort>              <compute answer; may abort>
  down(c2); // wait for answer              <put answer in b>
  <get answer from b; may abort>            up(c2);
}                                         }
```

Unfortunately, this solution does not work: any admissible execution requires an interleaved scheduling between the two transactions, thus violating isolation; hence, the transactions deadlock as none of them can progress. It is important to notice that this deadlock arises because interaction occurs between threads of *different* transactions; in fact, the solution above is perfectly fine for threads outside transactions or within the same transaction.

To overcome this limitation, in this paper we propose a programming model for *safe, data-driven* interactions between memory transactions. The key observation is that *atomicity* and *isolation* are two disjoint computational aspects:

- an *atomic non-isolated* block is executed “all-or-nothing”, but its execution can overlap others’ and *uncontrolled* access to shared data is allowed;
- a *non-atomic isolated* block is executed “as if it were the only one” (i.e., in mutual exclusion with others), but no rollback on errors is provided.

Thus, a “normal” block of code is neither atomic nor isolated; a mutex block (like Java *synchronized* methods) is isolated but not atomic; and a usual STM transaction is a block which is both atomic and isolated.

Our claim is that *atomic non-isolated blocks can be fruitfully used for implementing safe composable interacting memory transactions.* In this model, which we call *Atomic Transactional Memory* (ATM), a transaction is composed by several threads, called *participants*, which can cooperate on shared data. A transaction commits when all its participants commit, and aborts if any thread aborts. Threads participating to different transactions can access to shared data, but when this happens the transactions are *transparently merged* into a single one. For instance, the two transactions of the synchronization example above would automatically merge becoming the same transaction, so that the two threads can synchronize and proceed. Thus, this model relaxes the isolation requirement still guaranteeing atomicity and consistency; moreover, it allows for *loosely-coupled* interactions since transaction merging is driven only by run-time accesses to shared data, without any explicit coordination among the participants beforehand.

Related work. Many authors have proposed mechanisms to allow transactions to interact. Perhaps the work closest to ours are *transaction communicators* (TC) [9]. A transaction communicator is a (Java) object which can be accessed simultaneously by many transactions. To guarantee consistency, dependencies between transactions are maintained at runtime: a transaction can commit only when every transactions it depends on also commit. When dependencies form a cycle, the involved transactions must either all commit or abort together. This differs from ATM approach, where cooperating transactions are dynamically merged and hence the dependency graph is always acyclic; thus, ATM is opaque whereas TC is not. Other differences between TC and ATM are that our model has a formal semantics and that it can be implemented without changing neither the compiler nor the runtime (albeit it may be not very efficient).

Other authors have proposed *events*- and *message passing*-based mechanisms; we mention *transactional events* (TE) [1], which are specialized to the composition of send/receive operations to simplify synchronization in communication protocols, and TIC [20], where a transaction can be split into an isolated and a non-isolated transactions; this may violate local invariants and hence TIC does not satisfy opacity. Finally, *communicating memory transactions* (CMT) [8] is a model combining memory transactions with the actor model yielding *transactors*; hence CMT can be seen as the message-oriented counterpart of TC. CMT is opaque and has an efficient implementation; however it is best suited to distributed scenarios, whereas TC and ATM are aimed to multi-threaded programming on shared memory—in fact, transactors can be easily implemented in ATM by means of queues on shared memory. Another difference is that channel topology among transactors is established *a priori*, i.e. when the threads are created, while in ATM threads are created at runtime and interactions between transactions are driven by access to shared data only, whose references can be acquired at runtime.

Synopsis. In Section 2 we present *Atomic Transactional Memory* in the context of Concurrent Haskell. In Section 3 we provide a formal operational semantics. which is used in Section 4 to prove that ATM satisfies the *opacity* correctness criterion. Concluding remarks and directions for future work are in Section 5.

2 Haskell interface for Isolated and atomic transactions

In this section we give a brief overview of the interface for atomic transactions for Haskell. In fact, ATM can be implemented in any programming language, provided we have some means to forbid irreversible effects inside transactions; we have chosen Haskell because its typing system allows us to implement this restriction quite easily. Namely, we define two monads `ATM` and `ITM` (see Figure 1), representing the computational aspects of atomic *multi-threaded atomic*, non-isolated transactions and atomic *single-threaded isolated* transactions, respectively. Transactional memory locations are values of type `ATVar` and can be manipulated by isolated transactional actions only.

Using the construct `atomic`, programs in the `ATM` monad are executed "all-or-nothing" but without isolation; hence these transactions can merge at runtime.

```
-- Types for transactional actions ------------------------------------------
data ITM a  -- isolated atomic transactional action, return a value of type a
data ATM a  -- non-isolated atomic transaction, return a value of type a
-- Sequencing, do notation. Here t is a placeholder for ITM or ATM ----------
(>>=)  :: t a -> (a -> t b) -> t b
return :: a -> t a
-- Running isolated and atomic actions -------------------------------------
atomic   :: ATM a -> IO a    -- deliver the IO action for an atomic one
isolated :: ITM a -> ATM a   -- deliver the atomic action for an isolated one
retry    :: ITM a            -- retry the current transaction
orElse   :: ITM a -> ITM a -> ITM a -- fall back on the second action when
                                    -- the first action issues a retry
-- Exceptions ----------------------------------------------------------------
throw :: Exception e => e -> t a
catch :: Exception e => t a -> (e -> t a) -> t a
-- Threading -----------------------------------------------------------------
fork :: ATM () -> ATM ThreadId
-- Transactional shared memory -----------------------------------------------
data ATVar a            -- sharable memory location holding values of type a
newATVar     :: a -> ITM (ATVar a)
readATVar    :: ATVar a -> ITM a
writeATVar   :: ATVar a -> a -> ITM ()
```

Figure 1: The base interface of ATM.

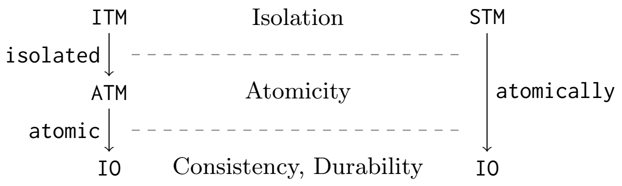


Figure 2: ACID computations: splitting `atomically`.

When needed, actions inside transactions can be executed in isolation by using the construct `isolated`. Both `ATM` and `ITM` transactions are *composable*; we exploit Haskell type system to prevent irreversible effects inside these monads. ATM is a conservative extension of STM [4]; in fact, STM's `atomically` is precisely the composition of `atomic` and `isolated` (Figure 2). This allows programmers to decide the granularity of interactions; e.g., the snippet below combines read and write actions to define an isolated atomic update of a transactional location.

```
modifyATVar :: ATVar a -> (a -> a) -> ITM ()
modifyATVar var f = do
  x <- readATVar var
  writeATVar var (f x)
```

Invariants on transactional locations can be easily checked by composing reads with checks that issue a retry if the invariant is not met, as in the snippet below.

```
assertATVar :: ATVar a -> (a -> Bool) -> ITM ()
assertATVar var p = do
  x <- readATVar var
  if (p x) then return () else retry
```

By sharing ATVars, non-isolated actions can share their view of transactional memory and affect each other. Consistency is guaranteed by merging transactions upon interaction thus the merged transaction may commit only if all participants agree on the final state of shared ATVars.

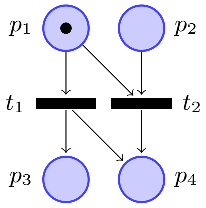
Example 2.1. Semaphores can be implemented in ATM as ATVars holding a counter and operation up and down as atomic and isolated actions. The first increments the counter held by the semaphore and the second decrements it when the counter is positive or retries the transaction otherwise thus blocking the thread. □

```
type Semaphore = ATVar Int
up :: Semaphore -> ITM ()
up s = modifyATVar s (1+)
down :: Semaphore -> ITM ()
down s = do
  assertATVar s (> 0)
  modifyATVar s (-1+)
```

Example 2.2. A Petri net is readily implemented in ATM by representing each transition by a thread and each place by a semaphore. Putting and taking a token from a place correspond to increasing (up) or decreasing (down) its semaphore—the latter blocks if no tokens are available. Each thread repeatedly simulates the firing of the transition it represents, by taking tokens from its input places and putting tokens in its output places. These semaphore operations must be performed atomically but not in isolation: isolation would prevent transitions sharing a place to fire concurrently. Using ATM, all this is achieved in few lines:

```
type Place = Semaphore
transition :: [Place] -> [Place] -> IO ThreadId
transition inputs outputs = forkIO (forever fire)
  where
    fire = atomic $ do
            mapM_ (isolated . down) inputs
            mapM_ (isolated . up)   outputs
```

Observe that, since firing is atomic but not isolated, this is an implementation of *true concurrent* Petri nets, which is usually more difficult to achieve than interleaving semantics. Here is a simple Petri net and its implementation:



```
main = do
  p1 <- atomically (newPlace 1)
  p2 <- atomically (newPlace 0)
  p3 <- atomically (newPlace 0)
  p4 <- atomically (newPlace 0)
  transition [p1] [p3, p4]
  transition [p1, p2] [p4]
```

When t_1 or t_2 race to take the token in p_1, if t_2 takes it first the transaction will eventually retry since there is no token in p_2. □

More examples can be found in [15].

3 Formal semantics of ATM

In this section we provide the formal semantics of ATM. Following [4], we fix an Haskell-like language extended with the ATM primitives of Figure 1 and characterise the behaviour of ATM by means of an abstract machine.

The language syntax is given by the following grammar:

$$\begin{aligned}
\textit{Values}\ \ V &::= r \mid \backslash x\ \texttt{->}\ M \mid \texttt{return}\ M \mid M\ \texttt{>>=}\ N \mid \texttt{throw}\ M \mid \texttt{catch}\ M\ N \\
&\mid \texttt{putChar}\ c \mid \texttt{getChar} \mid \texttt{fork}\ M \mid \texttt{atomic}\ M \mid \texttt{isolated}\ M \mid \texttt{retry} \\
&\mid M\ \texttt{`orElse`}\ N \mid \texttt{newATVar}\ M \mid \texttt{readATVar}\ r \mid \texttt{writeATVar}\ r\ M \\
\textit{Terms}\ \ M &::= x \mid V \mid M\,N
\end{aligned}$$

where the meta-variables x and r range over a given countable set of variables Var and of location names Loc, respectively. We assume Haskell typing conventions and denote the set of all well-typed terms by Term.

Terms are evaluated by an abstract state machine whose states are pairs $\langle P; \Sigma\rangle$ formed by: *(a)* a *thread family* (or *process*) $P = T_{t_1} \parallel \cdots \parallel T_{t_n}$ where t_i are unique *thread identifiers*; *(b)* a *memory* $\Sigma = \langle \Theta, \Delta, \Psi\rangle$, where $\Theta : \mathsf{Loc} \rightharpoonup \mathsf{Term}$ is the *heap* and $\Delta : \mathsf{Loc} \rightharpoonup \mathsf{Term} \times \mathsf{TrName}$ is the *working memory*; TrName is a set of names used to identify active transactions; Ψ is a forest of threads identifiers keeping track of how threads have been forked. Threads are the smaller unit of execution the machine scheduler operates on; they evaluate ATM terms and do not have any private transactional memory. A thread T_t has two forms: $(\!|M|\!)_t$ for threads evaluating a term M outside a transaction and $(\!|M; N|\!)_{t,k}$ for threads evaluating M inside transaction k with continuation N (the term to evaluate after that k has committed).

As for traditional closed (ACID) transactions (e.g., [4]), operations inside a transaction are evaluated against the distributed working memory Δ and effects are propagated to the heap Θ only on commits. When a thread inside a transaction k accesses a location outside Δ the location is *claimed by transaction* k and remains claimed until k commits, aborts or restarts. Threads in k can interact only with locations claimed by k, but active transactions can be merged to share their claimed locations. We denote the set of all possible states as State, and reference to each projected component of Σ by a subscript, i.e. Σ_Θ for the heap and Σ_Δ for the working memory. When describing updates to the memory Σ, we adopt the convention that Σ' has to be intended equals to Σ except if stated otherwise, i.e. by statements like $\Sigma'_\Theta = \Sigma_\Theta[r \mapsto M]$. Finally, $\varnothing$ denotes the empty heap and working memory.

Semantics The machine dynamics is defined by the two transition relations induced by the rules in Figures 3 to 6; auxiliary definitions are in Figure 7.

The first relation $M \to N$ is defined on terms only, and models pure computations (Figure 3). In particular, rule Eval allows a term M that is not a value to be evaluated by means of an auxiliary (partial) function $\mathcal{V}[M]$ yielding the value V; the other rules define the semantics of the monadic `bind` and exception handling in a standard way. It is interesting to notice the symmetry between

$$\frac{M \not\equiv V \quad \mathcal{V}[M] = V}{M \to V}\ \textsc{Eval}$$

$$\frac{}{\textsf{return}\ M\ \textsf{>>=}\ N \to N\,M}\ \textsc{BindVal} \qquad \frac{\textsf{e} \in \{\textsf{retry}, \textsf{throw}\ N\}}{\textsf{e}\ \textsf{>>=}\ M \to \textsf{e}}\ \textsc{BindEx}$$

$$\frac{\textsf{r} \in \{\textsf{retry}, \textsf{return}\ N\}}{\textsf{r}\ \textsf{`catch`}\ M \to \textsf{r}}\ \textsc{CatchVal} \qquad \frac{}{\textsf{throw}\ M\ \textsf{`catch`}\ N \to N\,M}\ \textsc{CatchEx}$$

Figure 3: Term reductions: $M \to N$.

$$\frac{}{\langle \mathbb{P}_t[\textsf{getChar}]; \Sigma\rangle \xrightarrow{?c} \langle \mathbb{P}_t[\textsf{return}\ c]; \Sigma\rangle}\ \textsc{InChar}$$

$$\frac{}{\langle \mathbb{P}_t[\textsf{putChar}\ c]; \Sigma\rangle \xrightarrow{!c} \langle \mathbb{P}_t[\textsf{return ()}]; \Sigma\rangle}\ \textsc{OutChar} \qquad \frac{M \to N}{\langle \mathbb{P}_t[M]; \Sigma\rangle \xrightarrow{\tau} \langle \mathbb{P}_t[N]; \Sigma\rangle}\ \textsc{TermIO}$$

$$\frac{t' \notin \textsf{threads}(\mathbb{P}_t[\textsf{fork}\ M])}{\langle \mathbb{P}_t[\textsf{fork}\ M]; \Sigma\rangle \xrightarrow{\tau} \langle \mathbb{P}_t[\textsf{return}\ t'] \parallel \llparenthesis M \rrparenthesis_{t'}; \Sigma\rangle}\ \textsc{ForkIO}$$

Figure 4: `IO` state transitions.

$$\frac{M \to N}{\langle \mathbb{T}_{t,k}[M]; \Sigma\rangle \xrightarrow{\tau} \langle \mathbb{T}_{t,k}[N]; \Sigma\rangle}\ \textsc{TermT}$$

$$\frac{t' \notin \textsf{threads}(\mathbb{T}_{t,k}[\textsf{fork}\ M]) \qquad \Sigma'_\Psi = \textsf{add\_child}(t, t', \Sigma_\Psi)}{\langle \mathbb{T}_{t,k}[\textsf{fork}\ M]; \Sigma\rangle \xrightarrow{\tau} \langle \mathbb{T}_{t,k}[\textsf{return}\ t'] \parallel \llparenthesis M; \textsf{return} \rrparenthesis_{t',k}; \Sigma'\rangle}\ \textsc{ForkT}$$

$$\frac{r \notin \mathrm{dom}(\Sigma_\Theta) \cup \mathrm{dom}(\Sigma_\Delta) \qquad \Sigma'_\Delta = \Sigma_\Delta[r \mapsto (M,k)]}{\langle \mathbb{T}_{t,k}[\textsf{newATVar}\ M]; \Sigma\rangle \xrightarrow{\tau} \langle \mathbb{T}_{t,k}[\textsf{return}\ r]; \Sigma'\rangle}\ \textsc{NewVar}$$

$$\frac{r \notin \mathrm{dom}(\Sigma_\Delta) \qquad \Sigma_\Theta(r) = M \qquad \Sigma'_\Delta = \Sigma_\Delta[r \mapsto (M,k)]}{\langle \mathbb{T}_{t,k}[\textsf{readATVar}\ r]; \Sigma\rangle \xrightarrow{\tau} \langle \mathbb{T}_{t,k}[\textsf{return}\ M]; \Sigma'\rangle}\ \textsc{Read1}$$

$$\frac{\Sigma_\Delta(r) = (M,j) \qquad \Sigma'_\Delta = \Sigma_\Delta[k \mapsto j]}{\langle \mathbb{T}_{t,k}[\textsf{readATVar}\ r]; \Sigma\rangle \xrightarrow{\tau} \langle \mathbb{T}_{t,j}[\textsf{return}\ M]; \Sigma'\rangle}\ \textsc{Read2}$$

$$\frac{r \notin \mathrm{dom}(\Sigma_\Delta) \qquad \Sigma'_\Delta = \Sigma_\Delta[r \mapsto (M,k)]}{\langle \mathbb{T}_{t,k}[\textsf{writeATVar}\ r\ M]; \Sigma\rangle \xrightarrow{\tau} \langle \mathbb{T}_{t,k}[\textsf{return ()}]; \Sigma'\rangle}\ \textsc{Write1}$$

$$\frac{\Sigma_\Delta(r) = (N,j) \qquad \Sigma'_\Delta = \Sigma_\Delta[k \mapsto j][r \mapsto (M,j)]}{\langle \mathbb{T}_{t,k}[\textsf{writeATVar}\ r\ M]; \Sigma\rangle \xrightarrow{\tau} \langle \mathbb{T}_{t,k}[\textsf{return ()}][k \mapsto j]; \Sigma'\rangle}\ \textsc{Write2}$$

$$\frac{\textsf{op} \in \{\textsf{throw}, \textsf{return}\} \qquad \langle \llparenthesis M; \textsf{return} \rrparenthesis_{t,k}; \Sigma\rangle \xrightarrow{\tau}{}^{*} \langle \llparenthesis \textsf{op}\ N; \textsf{return} \rrparenthesis_{t,j}; \Sigma'\rangle}{\langle \mathbb{T}_{t,k}[M\ \textsf{`orElse`}\ M']; \Sigma\rangle \xrightarrow{\tau} \langle \mathbb{T}_{t,j}[\textsf{op}\ N]; \Sigma'\rangle}\ \textsc{Or1}$$

$$\frac{\langle \llparenthesis M; \textsf{return} \rrparenthesis_{t,k}; \Sigma\rangle \xrightarrow{\tau}{}^{*} \langle \llparenthesis \textsf{retry}; \textsf{return} \rrparenthesis_{t,j}; \Sigma'\rangle}{\langle \mathbb{T}_{t,k}[M\ \textsf{`orElse`}\ M']; \Sigma\rangle \xrightarrow{\tau} \langle \mathbb{T}_{t,k}[M']; \Sigma\rangle}\ \textsc{Or2}$$

$$\frac{\textsf{op} \in \{\textsf{throw}, \textsf{return}\} \qquad \langle \llparenthesis M; \textsf{return} \rrparenthesis_{t,k}; \Sigma\rangle \xrightarrow{\tau}{}^{*} \langle \llparenthesis \textsf{op}\ N; \textsf{return} \rrparenthesis_{t,j}; \Sigma'\rangle}{\langle \mathbb{T}_{t,k}[\textsf{isolated}\ M]; \Sigma\rangle \xrightarrow{\tau} \langle \mathbb{T}_{t,j}[\textsf{op}\ N]; \Sigma'\rangle}\ \textsc{Isolated}$$

Figure 5: Transactional state transitions: $\langle P; \Sigma\rangle \xrightarrow{\tau} \langle P'; \Sigma'\rangle$.

$$\frac{}{\langle \llparenthesis \mathsf{atomic}\ M \ \texttt{>>=}\ N \rrparenthesis_t; \Sigma\rangle \xrightarrow{new\langle k\rangle} \langle \llparenthesis M; N \rrparenthesis_{t,k}; \Sigma\rangle}\ \textsc{New}$$

$$\frac{\Sigma'_\Theta = \mathsf{commit}(k,\Sigma) \qquad \Sigma'_\Delta = \mathsf{cleanup}(k,\Sigma)}{\langle \llparenthesis \mathsf{return}\ M; N \rrparenthesis_{t,k}; \Sigma\rangle \xrightarrow{co\langle k\rangle} \langle \llparenthesis \mathsf{return}\ M \ \texttt{>>=}\ N \rrparenthesis_t; \Sigma'\rangle}\ \textsc{Commit}$$

$$\frac{\Sigma'_\Theta = \mathsf{leak}(k,\Sigma) \qquad \Sigma'_\Delta = \mathsf{cleanup}(k,\Sigma) \qquad \Sigma'_\Psi = \mathsf{remove}(r,\Sigma_\Psi) \qquad r = \mathsf{root}(t,\Sigma_\Psi)}{\langle \llparenthesis \mathsf{throw}\ M; N \rrparenthesis_{t,k}; \Sigma\rangle \xrightarrow{ab\langle k,t,M\rangle} \langle \llparenthesis \mathsf{throw}\ M \ \texttt{>>=}\ N \rrparenthesis_t; \Sigma'\rangle}\ \textsc{Abort1}$$

$$\frac{\begin{array}{c} r = \mathsf{root}(t,\Sigma_\Psi) \qquad r = \mathsf{root}(t',\Sigma_\Psi) \\ \Sigma'_\Theta = \mathsf{leak}(k,\Sigma) \qquad \Sigma'_\Delta = \mathsf{cleanup}(k,\Sigma) \qquad \Sigma'_\Psi = \mathsf{remove}(r,\Sigma_\Psi) \end{array}}{\langle \llparenthesis M'; N \rrparenthesis_{t',k}; \Sigma\rangle \xrightarrow{\overline{ab}\langle k,t,M\rangle} \langle \llparenthesis \mathsf{throw}\ M \ \texttt{>>=}\ N \rrparenthesis_{t'}; \Sigma'\rangle}\ \textsc{Abort2}$$

$$\frac{\begin{array}{c} r = \mathsf{root}(t,\Sigma_\Psi) \qquad r \neq \mathsf{root}(t',\Sigma_\Psi) \\ \Sigma'_\Theta = \mathsf{leak}(k,\Sigma) \qquad \Sigma'_\Delta = \mathsf{cleanup}(k,\Sigma) \qquad \Sigma'_\Psi = \mathsf{remove}(r,\Sigma_\Psi) \end{array}}{\langle \llparenthesis M'; N \rrparenthesis_{t',k}; \Sigma\rangle \xrightarrow{\overline{ab}\langle k,t,M\rangle} \langle \llparenthesis \mathsf{retry} \rrparenthesis_{t'}; \Sigma'\rangle}\ \textsc{Abort3}$$

$$\frac{\langle P;\Sigma\rangle \xrightarrow{ab\langle k,t,M\rangle} \langle P';\Sigma'\rangle \qquad \langle Q;\Sigma\rangle \xrightarrow{\overline{ab}\langle k,t,M\rangle} \langle Q';\Sigma'\rangle}{\langle P \parallel Q;\Sigma\rangle \xrightarrow{ab\langle k,t,M\rangle} \langle P' \parallel Q';\Sigma'\rangle}\ \textsc{MCastAb}$$

$$\frac{\langle P;\Sigma\rangle \xrightarrow{co\langle k\rangle} \langle P';\Sigma'\rangle \qquad \langle Q;\Sigma\rangle \xrightarrow{co\langle k\rangle} \langle Q';\Sigma'\rangle}{\langle P \parallel Q;\Sigma\rangle \xrightarrow{co\langle k\rangle} \langle P' \parallel Q';\Sigma'\rangle}\ \textsc{MCastCo}$$

$$\frac{\langle P;\Sigma\rangle \xrightarrow{\beta} \langle P';\Sigma'\rangle \qquad \beta \neq \tau \qquad \mathsf{transaction}(\beta) \notin \mathsf{transactions}(Q)}{\langle P \parallel Q;\Sigma\rangle \xrightarrow{\beta} \langle P' \parallel Q;\Sigma'\rangle}\ \textsc{MCastGroup}$$

Figure 6: Transaction management transitions: $\langle P;\Sigma\rangle \xrightarrow{\beta} \langle P';\Sigma'\rangle$.

`bind` and `catch` and how `retry` is treated as an exception by rule BindEx and as a result value by rule CatchVal.

Relation $\to$ is used to define the labelled transition relation $\langle P;\Sigma\rangle \xrightarrow{\beta} \langle P';\Sigma'\rangle$ over states. This relation is non deterministic, to model the fact that the scheduler can choose among various threads to execute next; therefore, several rules can apply to a given state according to different evaluation contexts:

$$\begin{array}{rlrl} \textit{Expression:} & \mathbb{E} ::= [-] \mid \mathbb{E} \ \texttt{>>=}\ M & \textit{Plain process:} & \mathbb{P}_t ::= \llparenthesis \mathbb{E} \rrparenthesis_t \parallel P \\ \textit{Transaction:} & \mathbb{T}_{t,k} ::= \llparenthesis \mathbb{E}; M \rrparenthesis_{t,k} \parallel P & \textit{Any process:} & \mathbb{A}_t ::= \mathbb{P}_t \mid \mathbb{T}_{t,k} \end{array}$$

Labels β describe the kind of transition, and are defined as follows:

$$\beta ::= \tau \mid new\langle k\rangle \mid co\langle k\rangle \mid ab\langle k,t,M\rangle \mid \overline{ab}\langle k,t,M\rangle \mid ?c \mid !c$$

where $k \in \mathsf{TrName}$, $M \in \mathsf{Term}$ as usual.

Transitions labelled by τ represent *internal* steps of transactions, i.e., which do not need any coordination: reduction of pure terms, memory operations and

$$
\begin{aligned}
\mathsf{threads}(T_{t_1} \parallel \cdots \parallel T_{t_n}) &\triangleq \{t_1, \dots t_n\}\\
\mathsf{transaction}(\beta) &\triangleq k \text{ for } \beta \in \{new\langle k\rangle, co\langle k\rangle, ab\langle k,t,M\rangle, \overline{ab}\langle k,t,M\rangle\}\\
(\Delta[k \mapsto j])(r) &\triangleq \begin{cases} \Delta(r) & \text{if } \Delta(r) = (M,l), l \neq k\\ (M,j) & \text{if } \Delta(r) = (M,k) \end{cases}\\
\mathsf{transactions}(P) &\triangleq \begin{cases} \mathsf{transactions}(P_1) \cup \mathsf{transactions}(P_2) & \text{if } P = P_1 \parallel P_2\\ \{k\} & \text{if } P = (\!| M; N |\!)_{t,k}\\ \emptyset & \text{otherwise} \end{cases}\\
P[k \mapsto j] &\triangleq \begin{cases} P_1[k \mapsto j] \parallel P_2[k \mapsto j] & \text{if } P = P_1 \parallel P_2\\ (\!| M; N |\!)_{t,j} & \text{if } P = (\!| M; N |\!)_{t,k}\\ P & \text{otherwise} \end{cases}\\
\Theta[r \mapsto M](s) &\triangleq \text{if } r = s \text{ then } M \text{ else } \Theta(s)\\
\Delta[r \mapsto (M,k)](s) &\triangleq \text{if } r = s \text{ then } (M,k) \text{ else } \Delta(s)\\
\mathsf{cleanup}(k, \Sigma)(r) &\triangleq \text{if } \Sigma_\Delta(r) = (M,k) \text{ then } \perp \text{ else } \Sigma_\Delta(r)\\
\mathsf{commit}(k, \Sigma)(r) &\triangleq \text{if } \Sigma_\Delta(r) = (M,k) \text{ then } M \text{ else } \Sigma_\Theta(r)\\
\mathsf{leak}(k, \Sigma)(r) &\triangleq M \text{ if } \Sigma_\Theta(r) = M \text{ or } \Sigma_\Theta(r) = \perp \text{ and } \Sigma_\Delta(r) = (M,k)
\end{aligned}
$$

Figure 7: Auxiliary functions used by the formal semantics of ATM.

thread creation (see rules in Figure 5). Reading a location falls into two cases: rule Read1 models the reading of an unclaimed location and its effect is to record the claim in Δ, while rule Read2 models the reading of a claimed location and its effect is to merge the transactions of the current thread with that claiming the location. Writes behave similarly. Rules Or1 and Or2 describe the semantics of alternative sub-transactions: if the first one `retry`-es the second is executed discarding any effect of the first. Rule ForkT spawns a new thread for the current transaction; a term `fork` M can appear inside `atomic`, thus allowing multi-threaded atomic transactions, but its use inside `isolated` is prevented by the type system and by the shape of rule Isolated as well.

The remaining labels describe state transitions concerning the life-cycle of transactions: creation, commit, abort, and restart (see rules in Figure 6). These operations require a coordination among threads; for instance, an abort from a thread has to be propagated to every thread participating to the same transaction. This is captured in the semantics by labelling the transition with the operation and the name of the transaction involved; this information is used to force synchronisation of all participants of that transaction. To illustrate this mechanism, we describe the commit of a transaction k, namely $\langle P; \Sigma\rangle \xrightarrow{co\langle k\rangle} \langle P'; \Sigma'\rangle$. First, by means of rule MCastGroup we split P into two subprocesses, one of which contains all threads participating in k (those not in k cannot do a transition whose label contains k). Secondly, using recursively rule MCastCo we single out every thread in k. Finally, we apply rule Commit provided that every thread is ready to commit, i.e., it is of the form $(\!| \texttt{return}\ M; N |\!)_{t,k}$.

Aborting a transaction works similarly, but it based on vetoes instead of an unanimous vote. Aborts are triggered by unhandled exceptions raised by some thread, but threads react to this situation in different ways:

- threads forked within the transaction, in the same tree of the thread raising the exception: these threads are killed (and the root thread aborted) because their creation must be discarded, as for any transactional side-effect;
- threads from different trees which joined the transaction after it was created, due to a merging: these threads just retry their transaction, since aborting would require them to handle exceptions raised by "foreign" threads.

Like Haskell STM [4], aborts leak some effects namely any transactional variable created in the aborted transaction that also occurs in the aborting exception.

Note that there are no derivation rules for `retry`: its meaning is to inform the scheduler that we have reached a state where the execution is stuck; hence the machine has to re-execute the transaction from the beginning (or backtracking from a suitable check-point), possibly following a different execution order.

4 Opacity

In this section we validate the formal semantics of ATM by proving it satisfies the *opacity* correctness criterion for transactional memory [3]. This criterion is an extension of the classical *serialisability property* for databases with the additional requirement that even non-committed transactions must access consistent states. Intuitively, this property ensures that *(a)* effects of any committed transaction appear performed at a single, indivisible point during the transaction lifetime; *(b)* updates of any aborted transaction cannot be seen by other transactions; *(c)* transactions always access consistent states of the system.

In order to formally capture these intuitive requirements let us recall some notions from [3]. A *history* is a sequence of `read`, `write`, `commit`, and `abort` operations ordered according to the time at which they were issued (simultaneous events are arbitrarily ordered) and such that no operation can be issued by a transaction that has already performed a `commit` or an `abort`. A transaction k is said to be in a history H if the latter contains at least one operation issued by k. Histories that differ only for the relative position of operations in different transactions are considered *equivalent*. Any history H defines a *happens-before* partial order $\prec_H$ over transactions, where $k \prec_H k'$ iff the transaction k becomes committed or aborted in H before k' issues its first operation. If $\prec_H$ is total then H is called *sequential*. For a history H, let $complete(H)$ be the set of histories obtained by adding either a commit or an abort for every live transaction in H.

We can now recall Guerraoui-Kapałka's definition[3] of opacity [3, Def. 1].

Definition 4.1 (Opacity). *A history H is said to be* opaque *if there exists a sequential history S equivalent to some history in $complete(H)$ such that S preserves the happens-before order of H.*

[3] The original definition requires the history H to be "legal", but this notion is relevant only in presence of non-transactional operations which ATM prevents by design.

As shown in [3], opacity corresponds to the absence of mutual dependencies between live transactions, where a dependency is created whenever a transaction reads an information written by another or depends from its outcome.

Definition 4.2 (Opacity graph [3, Sec. 5.4]). *For a history H let $\ll$ be a total order on the set T of all transactions in H. An* opacity graph $OPG(H, \ll)$ *is a bi-coloured directed graph on T such that a vertex is* red *if the corresponding transaction is either running or aborted, it is* black *otherwise, and for all vertices $k, k' \in T$, there is a edge $k \longrightarrow k'$ if any of the following holds:*

1. *k' happens-before k ($k' \prec_H k$);*
2. *k reads something written by k';*
3. *k' reads some location written by k and $k' \ll k$;*
4. *k' is neither running nor aborted and there are a location r and a transaction k'' such that $k' \ll k''$, k' writes to r, and k'' reads r from k.*

The edge is red *if the second case applies, otherwise it is black. The graph is said to be* well-formed *if all edges from red nodes in $OPG(H, \ll)$ are also red.*

Let H be a history and let k be a transaction appearing in it. A `read` operation by k is said to be *local* (to k) whenever the previous operation by k on the same location was a `write`. A `write` operation by k is said to be *local* (to k) whenever the next operation by k on the same location is a `write`. We denote by *nonlocal*(H) the longest sub-history of H without any local operations. A history H is said *locally-consistent* if every local `read` is preceded by a `write` operation that writes the read value; it is said *consistent* if, additionally, whenever some k reads v from r in *nonlocal*(H) then some k' writes v to r in *nonlocal*(H).

Theorem 4.1 ([3, Thm. 2]). *A history H is opaque if and only if (a) H is consistent and (b) there exists a total order $\ll$ on the set of transactions in H such that $OPG(nonlocal(H), \ll)$ is well-formed and acyclic.*

In [3] transactions may encapsulate several threads but cannot be merged. Therefore, in order to study opacity of ATM we extend the set of operations considered in *loc. cit.* with explicit merges. Let k, k' be two running transactions in the given history; when they merge, they share their threads, locations, and effects. From this perspective, k is commit-pending and depends from k' and hence in the opacity graph, k is a red node connected to k' by a red edge. Hence, merges can be equivalently expressed at the history level by sequences like:

(1) new x; *(2)* k' writes on x; *(3)* k reads from x; *(4)* k prepares to commit.

These are the only dependencies found in histories generated by ATM.

Theorem 4.2. *For H a history describing an execution of a* ATM *program and a total order $\ll$, $OPG(nonlocal(H), \ll)$ is a forest of red edges where only roots may be black.*

Proof. By inspection of the rules it is easy to see that *(a)* transactions may access only locations they claimed; *(b)* claimed locations are released only on `commits`, aborts and retries; *(c)* transactions have to merge with any transaction holding a location they need. Therefore, at any given time there is at most one running transaction issuing operations on a given location, hence reads and writes do not create edges. Thus edges are created only during the execution of merges and, by inspecting the above implementation, it easy to see that *(d)* any transaction can issue at most one merge; *(e)* a transaction issuing a merge is a red node; *(f)* the edge created by a merge is red. Therefore, transactions form a forest made of red edges where any non-root node is red. □

Since a forest formed by red edges whose sources are always red is always acyclic and well-formed, we can conclude our correctness result:

Corollary 4.1 (Opacity). ATM *meets the opacity criterion.*

5 Conclusions and future work

In this paper we have presented ATM, a programming model supporting interactions between composable memory transactions. This model separates isolated transactions from non-isolated ones, still guaranteeing atomicity; the latter can interact by accessing to shared variables. Consistency is ensured by transparently *merging* interacting transactions at runtime. We have given a formal semantics for ATM, and proved that this model satisfies the important *opacity* criterion.

As future work, it would be interesting to add some heuristics to better handle retry events. Currently, a retry restarts all threads participating to the transaction; a more efficient implementation would keep track of the *working set* of each thread, and at a retry we need to restart only the threads whose working sets have non-empty intersection with that being restarted. Another optimization is to implement transactions and ATVars directly in the runtime, akin the implementation of STM in the Glasgow Haskell Compiler [4].

We have presented ATM within Haskell (especially to leverage its type system), but this model is general and can be applied to other languages. A possible future work is to port this model to an imperative object oriented language, such as Java or C++; however, like other TM implementations, we expect that this extension will require some changes in the compiler and/or the runtime.

This work builds on the calculus with shared memory and atomic transactions described in [16]. In *op. cit.* this model is shown to be expressive enough to represent $TCCS^m$ [7], a variant of the Calculus of Communicating Systems with transactional synchronization. The relation is strict since there are no sensible ways to represent in $TCCS^m$ features like unbounded memory allocation, aliasing and higher-order computations. The lack of such correspondence calls for an extension of $TCCS^m$ with name mobility and restriction, *i.e.* a variant of the π-calculus with transactional communication. Close to this line of investigation is the study of interacting transactions in the setting of [14]. Being based on CCS, communication in $TCCS^m$ is synchronous; however, nowadays asynchronous models play an important rôle (see actors, event-driven programming, etc.), so it may be interesting to generalize the model to consider also this case, e.g. by defining a calculus for event-driven models or an actor-based calculus with atomic transactions. Such a calculus can be quite useful also for modelling speculative reasoning for cooperating systems [10–13] or study distributed interacting transactions in serverless-computing [2, 6, 18]. A local version of actor-based atomic transactions can be implemented in ATM using lock-free data structures (e.g., message queues) in shared transactional memory.

Bibliography

[1] K. Donnelly and M. Fluet. Transactional events. *J. Funct. Program.*, 18 (5-6):649–706, 2008.

[2] M. Gabbrielli, S. Giallorenzo, I. Lanese, F. Montesi, M. Peressotti, and S. P. Zingaro. No more, no less - A formal model for serverless computing. In *COORDINATION*, volume 11533 of *Lecture Notes in Computer Science*, pages 148–157. Springer, 2019.

[3] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM.

[4] T. Harris, S. Marlow, S. L. Peyton Jones, and M. Herlihy. Composable memory transactions. In *Proc. PPOPP*, pages 48–60, 2005.

[5] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In A. J. Smith, editor, *Proceedings of the 20th Annual International Symposium on Computer Architecture. San Diego, CA, May 1993*, pages 289–300. ACM, 1993.

[6] A. Jangda, D. Pinckney, Y. Brun, and A. Guha. Formal foundations of serverless computing. *Proc. ACM Program. Lang.*, 3(OOPSLA):149:1–149:26, 2019.

[7] V. Koutavas, C. Spaccasassi, and M. Hennessy. Bisimulations for communicating transactions - (extended abstract). In A. Muscholl, editor, *Foundations of Software Science and Computation Structures - 17th International Conference, FOSSACS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8412 of *Lecture Notes in Computer Science*, pages 320–334. Springer, 2014.

[8] M. Lesani and J. Palsberg. Communicating memory transactions. In C. Cascaval and P. Yew, editors, *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, pages 157–168. ACM, 2011.

[9] V. Luchangco and V. J. Marathe. Transaction communicators: Enabling cooperation among concurrent transactions. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 169–178, New York, NY, USA, 2011. ACM.

[10] J. Ma, K. Broda, R. Goebel, H. Hosobe, A. Russo, and K. Satoh. Speculative abductive reasoning for hierarchical agent systems. In J. Dix, J. Leite,

G. Governatori, and W. Jamroga, editors, *Computational Logic in Multi-Agent Systems*, volume 6245 of *Lecture Notes in Computer Science*, pages 49–64. Springer Berlin Heidelberg, 2010.

[11] A. Mansutti, M. Miculan, and M. Peressotti. Multi-agent systems design and prototyping with bigraphical reactive systems. In K. Magoutis and P. Pietzuch, editors, *Proc. DAIS*, volume 8460 of *Lecture Notes in Computer Science*, pages 201–208. Springer, 2014.

[12] A. Mansutti, M. Miculan, and M. Peressotti. Distributed execution of bigraphical reactive systems. *ECEASST*, 71, 2014.

[13] A. Mansutti, M. Miculan, and M. Peressotti. Towards distributed bigraphical reactive systems. In R. Echahed, A. Habel, and M. Mosbah, editors, *Proc. GCM'14*, page 45, 2014. Workshop version.

[14] D. Medic, C. A. Mezzina, I. Phillips, and N. Yoshida. Towards a formal account for software transactional memory. In *RC*, volume 12227 of *Lecture Notes in Computer Science*, pages 255–263. Springer, 2020.

[15] M. Miculan and M. Peressotti. Software transactional memory with interactions. *CoRR*, abs/2007.10809, 2020.

[16] M. Miculan, M. Peressotti, and A. Toneguzzo. Open transactions on shared memory. In *COORDINATION*, volume 9037 of *Lecture Notes in Computer Science*, pages 213–229. Springer, 2015.

[17] Y. Ni, V. Menon, A. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In K. A. Yelick and J. M. Mellor-Crummey, editors, *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007, San Jose, California, USA, March 14-17, 2007*, pages 68–78. ACM, 2007.

[18] M. Obetz, A. Das, T. Castiglia, S. Patterson, and A. Milanova. Formalizing event-driven behavior of serverless applications. In *ESOCC*, volume 12054 of *Lecture Notes in Computer Science*, pages 19–29. Springer, 2020.

[19] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[20] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with isolation and cooperation. In R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 191–210. ACM, 2007.