Open Transactions on Shared Memory

Marino Miculan, Marco Peressotti, Andrea Toneguzzo

Open transactions on shared memory

Marino Miculan, Marco Peressotti, and Andrea Toneguzzo

Laboratory of Models and Applications of Distributed Systems
Department of Mathematics and Computer Science, University of Udine
{marino.miculan, marco.peressotti}@uniud.it

Abstract. *Transactional memory* has arisen as a good way for solving many of the issues of lock-based programming. However, most implementations admit *isolated* transactions only, which are not adequate when we have to coordinate *communicating* processes. To this end, in this paper we present *OCTM*, an Haskell-like language with *open* transactions over shared transactional memory: processes can *join* transactions at runtime just by accessing to shared variables. Thus a transaction can co-operate with the environment through shared variables, but if it is rolled-back, also all its effects on the environment are retracted. For proving the expressive power of *OCTM* we give an implementation of $TCCS^m$, a CCS-like calculus with open transactions.

1 Introduction

Coordination of concurrent programs is notoriously difficult. Traditional fine-grained lock-based mechanisms are deadlock-prone, inefficient, not composable and not scalable. For these reasons, *Software Transactional Memory* (STM) has been proposed as a more effective abstraction for concurrent programming [1,9,17]. The idea is to mark blocks of code as "atomic"; at runtime, these blocks are executed so that the well-known ACID properties are guaranteed. Transactions ensure deadlock freedom, no priority inversion, automatic roll-back on exceptions or timeouts, and greater parallelizability. Among other implementations, we mention *STM Haskell* [7], which allows atomic blocks to be composed into larger ones. STM Haskell adopts an *optimistic* evaluation strategy: the blocks are allowed to run concurrently, and eventually if an interference is detected a transaction is *aborted* and its effects on the memory are rolled back.

However, standard ACID transactions are still inadequate when we have to deal with *communicating* processes, i.e., which can exchange information *during* the transactions. This is very common in concurrent distributed programming, like in service-oriented architectures, where processes dynamically combine to form a transaction, and all have to either commit or abort together. In this scenario the participants cannot be enclosed in one transaction beforehand, because transactions are formed at runtime. To circumvent this issue, various forms of *open transactions* have been proposed, where the Isolation requirement is relaxed [2–4,10,12]. In particular, *TransCCS* and $TCCS^m$ are two CCS-like calculi recently introduced to model communicating transactions [4, 5, 10]. These calculi offer methodologies for proving important properties, such as fair-testing for proving liveness and bisimulations for proving contextual equivalences.

Now, if we try to implement cross-transaction communications *a la* $TCCS^m$ in STM Haskell or similar languages, it turns out that isolated transactions are not expressive enough. As an example, let us consider two $TCCS^m$ transactions $\langle \bar{c}.P \blacktriangleright 0 \rangle | \langle c.Q \blacktriangleright 0 \rangle$ synchronizing on a channel c. Following the standard practice, we could implement this synchronization as two parallel processes using a pair of semaphores `c1,c2` (which are easily realized in STM Haskell):

```
⟨c̄.P ▶ 0⟩ = atomic {          ⟨c.Q ▶ 0⟩ = atomic {
               up c1     -- 1.1              down c1   -- 2.1
               down c2   -- 1.2              up c2     -- 2.2
               P                             Q
             }                           }
```

This implementation is going to deadlock: the only possible execution order is 1.1-2.1-2.2-1.2, which is possible outside transactions but it is forbidden for ACID transactions[1]. The problem is that ordinary STM transactions are kept isolated, while in $TCCS^m$ they can merge at runtime.

In order to address this issue, in this paper we introduce software transactional memory with *open* transactions: processes can *join* transactions and transactions can *merge* at runtime, when they access to shared variables. To this end, we present *OCTM*, a higher-order language extending the concurrency model of STM Haskell with composable *open (multi-thread)* transactions interacting via *shared memory*. The key step is to separate the isolation aspect from atomicity: in *OCTM* the `atomic` construct ensures "all-or-nothing" execution, but not isolation; when needed, isolated execution can be guaranteed by a new constructor `isolated`. An `atomic` block is a *participant* (possibly the only one) of a transaction. Notice that transaction merging is implicitly triggered by accessing to shared memory, without any explicit operation or *a priori* coordination. For instance, in *OCTM* the two transactions of the example above would merge becoming two participants of the same transaction, hence the two threads can synchronize and proceed. In order to prove formally the expressivity of open memory transactions, we define an implementation of $TCCS^m$ in *OCTM*, which is proved to correctly preserve behaviours by means of a suitable notion of simulation. We have based our work on STM Haskell as a paradigmatic example, but this approach is general and can be applied to other STM implementations.

Lesani and Palsberg [12] have proposed transactions communicating through transactional message-based channels called *transactional events*. These mechanisms are closer to models like TransCCS and $TCCS^m$, but on the other hand they induce a *strict coupling* between processes, which sometimes is neither advisable nor easy to implement (e.g., when we do not know all transaction's participants beforehand). In fact, most STM implementations (including STM Haskell) adopt the shared memory model of multi-thread programming; this model is also more amenable to implementation on modern multi-core hardware architectures with transactional memory [8]. For these reasons, in *OCTM* we have preferred to stick to *loosely coupled* interactions based on shared memory only.

[1] This possibility was pointed out also in [7]: "two threads can easily deadlock if each awaits some communication from the other".

$$
\begin{array}{lll}
\text{Value} & V ::= & r \mid \lambda x.M \mid \texttt{return}\ M \mid M\ \texttt{>>=}\ N \mid \\
& & \texttt{newVar}\ M \mid \texttt{readVar}\ r \mid \texttt{writeVar}\ r\ M \mid \\
& & \texttt{fork}\ M \mid \texttt{atomic}\ M\ N \mid \texttt{isolated}\ M \mid \texttt{abort}\ M \mid \texttt{retry} \\
\text{Term} & M, N ::= & x \mid V \mid MN \mid \ldots
\end{array}
$$

Fig. 1. Syntax of *OCTM* values and terms.

The rest of the paper is structured as follows. In Section 2 we describe the syntax and semantics of *OCTM*. The calculus $TCCS^m$, our reference model for open transactions, is recalled in Section 3. Then, in Section 4 we provide an encoding of $TCCS^m$ in *OCTM*, proving that *OCTM* is expressive enough to cover open transactions. Conclusions and directions for future work are in Section 5. Longer proofs are in the extended version of this paper [15].

2 *OCTM*: Open Concurrent Transactional Memory

In this section we introduce the syntax and semantics of *OCTM*, a higher-order functional language with threads and open transaction on shared memory. The syntax is Haskell-like (in the wake of existing works on software transactional memories such as [7]) and the semantics is a small-step operational semantics given by two relations: $\xrightarrow{\beta}$ models transaction auxiliary operations (e.g. creation) while $\rightarrow$ models actual term evaluations. Executions proceeds by repeatedly choosing a thread and executing a single (optionally transactional) operation; transitions from different threads may be arbitrarily interleaved as long as atomicity and isolation are not violated where imposed by the program.

2.1 Syntax

The syntax can be found in Figure 1 where the meta-variables r and x range over a given countable set of locations Loc and variables Var respectively. Terms and values are inspired to Haskell and are entirely conventional[2]; they include abstractions, application, monadic operators (`return` and `>>=`), memory operators (`newVar`, `readVar`, `writeVar`), forks, transactional execution modalities (`atomic` and `isolated`) and transaction operators (`abort` and `retry`).

Effectfull expressions such as `fork` or `isolated` are glued together by the (overloaded) monadic bind `>>=` e.g.:

$$\texttt{newVar 0 >>=}\ \lambda x.\texttt{(fork (writeVar}\ x\ \texttt{42) >>=}\ \lambda y.\texttt{readVar}\ x\texttt{)}$$

whereas values are "passed on" by the monadic unit `return`.

Akin to Haskell, we will use underscores in place of unused variables (e.g. $\lambda\_.\texttt{0}$) and $M \gg N$ as a shorthand for $M\ \texttt{>>=}\ \lambda\_.N$, and the convenient `do`-*notation*:

$$
\begin{aligned}
\texttt{do}\{x \leftarrow M; N\} &\equiv M\ \texttt{>>=}\ (\lambda x.\texttt{do}\{N\}) \\
\texttt{do}\{M; N\} &\equiv M\ \texttt{>>=}\ (\lambda\_.\texttt{do}\{N\}) \\
\texttt{do}\{M\} &\equiv M
\end{aligned}
$$

[2] Although we treat the application of monadic combinators (e.g. `return`) as values in the line of similar works [7].

possibly trading semicolons and brackets for the conventional Haskell *layout*. For instance, the above example is rendered as

```
do
   x ← newVar 0
   fork (writeVar x 42)
   readVar x
```

2.2 Operational Semantics

We present the operational semantics of *OCTM* in terms of an abstract machine whose states are triples $\langle P; \Theta, \Delta \rangle$ formed by

- thread family (process) P;
- heap memory $\Theta : \mathsf{Loc} \rightharpoonup \mathsf{Term}$;
- distributed working memory $\Delta : \mathsf{Loc} \rightharpoonup \mathsf{Term} \times \mathsf{TrName}$

where Term denotes the set of *OCTM* terms (cf. Figure 1) and TrName denotes the set of names used by the machine to identify active transactions. We shall denote the set of all possible states as State.

Threads Threads are the smaller unit of execution the machine scheduler operates on; they execute OCTM terms and do not have any private transactional memory. Threads are given unique identifiers (ranged over by t or variations thereof) and, whenever they take part to some transaction, the transaction identifier (ranged over k, j or variations thereof). Threads of the former case are represented by $(\!|M|\!)_t$ where M is the term being evaluated and the subscript t is the thread identifier. Threads of the latter case have two forms: $(\!|M \triangleright M'; N|\!)_{t,k}$, called and $(\!|M \triangleright M'|\!)_{t,k}$ where:

- M is the term being evaluated inside the transaction k;
- M' is the term being evaluated as *compensation* in case k is aborted;
- N is the term being evaluated as *continuation* after k commits or aborts.

Threads with a continuation are called *primary participants (to transaction k)*, while threads without continuation are the *secondary participants*. The former group includes all and only the threads that started a transaction (i.e. those evaluated in an `atomic`), while the latter group encompasses threads forked inside a transaction and threads forced to join a transaction (from outside a transactional context) because of memory interactions. While threads of both groups can force a transaction to abort or restart, only primary participants can vote for its commit and hence pass the transaction result to the continuation.

We shall present thread families using the evocative CCS-like parallel operator $\|$ (cf. Figure 2) which is commutative and associative. Notice that this operator is well-defined only on operands whose thread identifiers are distinct. The notation is extended to thread families with $\mathbf{0}$ denoting the empty family.

$$
\begin{array}{lll}
\text{Thread} & T_t ::= (\!|M|\!)_t \mid (\!|M \rhd M'; N|\!)_{t,k} \mid (\!|M \rhd M'|\!)_{t,k} \\
\text{Thread family} & P ::= T_{t_1} \parallel \cdots \parallel T_{t_n} \quad \forall i, j\ t_i \neq t_j \\
\text{Expressions} & \mathbb{E} ::= [-] \mid \mathbb{E} \mathbin{\texttt{>>=}} M \\
\text{Processes} & \mathbb{P}_t ::= (\!|\mathbb{E}|\!)_t \\
\text{Transactions} & \mathbb{T}_{t,k} ::= (\!|\mathbb{E} \rhd M; N|\!)_{t,k} \mid (\!|\mathbb{E} \rhd M|\!)_{t,k}
\end{array}
$$

Fig. 2. Threads and evaluation contexts.

$$
\frac{M \not\equiv V \quad \mathcal{V}[M] = V}{M \to V}\ (\textsc{Eval}) \qquad \frac{}{\texttt{return}\ M \mathbin{\texttt{>>=}} N \to NM}\ (\textsc{BindReturn})
$$

$$
\frac{}{\texttt{retry} \mathbin{\texttt{>>=}} M \to \texttt{retry}}\ (\textsc{BindRetry}) \qquad \frac{}{\texttt{abort}\ N \mathbin{\texttt{>>=}} M \to \texttt{abort}\ N}\ (\textsc{BindAbort})
$$

Fig. 3. *OCTM* semantics: rules for term evaluation.

Memory The memory is divided in the heap Θ and in a distributed working memory Δ. As for traditional closed (acid) transactions (e.g. [7]), operations inside a transaction are evaluated against Δ and effects are propagated to Θ only on commits. When a thread inside a transaction k accesses a location outside Δ the location is *claimed for* k and remains claimed for the rest of k execution. Threads inside a transaction can interact only with locations claimed by their transaction. To this end, threads outside any transaction can join an existing one and different active transactions can be merged to share their claimed locations.

We shall denote the pair $\langle \Theta, \Delta \rangle$ by Σ and reference to each projected component by a subscript e.g. Σ_Θ for the heap. When describing updates to the state Σ, we adopt the convention that Σ' has to be intended as equal to Σ except if stated otherwise, i.e. by statements like $\Sigma'_\Theta = \Sigma_\Theta[r \mapsto M]$. Formally, updates to location content are defined on Θ and Δ as follows:

$$
\Theta[r \mapsto M](s) \triangleq \begin{cases} M & \text{if } r = s \\ \Theta(s) & \text{otherwise} \end{cases} \qquad \Delta[r \mapsto (M,k)](s) \triangleq \begin{cases} (M,k) & \text{if } r = s \\ \Delta(s) & \text{otherwise} \end{cases}
$$

for any $r, s \in \mathsf{Loc}$, $M \in \mathsf{Term}$ and $k \in \mathsf{TrName}$. Likewise, updates on transaction names are defined on Σ and Δ as follows:

$$
\Sigma[k \mapsto j] \triangleq (\Theta, \Delta[k \mapsto j]) \quad (\Delta[k \mapsto j])(r) \triangleq \begin{cases} \Delta(r) & \text{if } \Delta(r) = (M,l), l \neq k \\ (M,j) & \text{if } \Delta(r) = (M,k) \end{cases}
$$

for any $r \in \mathsf{Loc}$, $M \in \mathsf{Term}$ and $k, j \in \mathsf{TrName}$. Note that j may occur in Δ resulting in the fusion of the transactions denoted by k and j respectively. Finally, $\varnothing$ denotes the empty memory (i.e. the completely undefined partial function).

Behaviour Evaluation contexts are shown in Figure 2 and the transition relations are presented in Figures 3, 4, 5. The first (cf. Figures 3) is defined on terms only and models pure computations. In particular, rule (Eval) allows a term M that is not a value to be evaluated by an auxiliary (partial) function, $\mathcal{V}[M]$ yielding the value V of M whereas the other three rules define the semantic of the monadic bind. The transition relation modelling pure computations can be thought as accessory to the remaining two for these model transitions between the states of the machine under definition.

$$\frac{M \to N}{\langle \mathbb{P}_t[M] \| P; \Sigma\rangle \to \langle \mathbb{P}_t[N] \| P; \Sigma\rangle}\ (\textsc{TermP}) \qquad \frac{M \to N}{\langle \mathbb{T}_{t,k}[M] \| P; \Sigma\rangle \to \langle \mathbb{T}_{t,k}[N] \| P; \Sigma\rangle}\ (\textsc{TermT})$$

$$\frac{t' \notin \mathsf{threads}(P) \quad t \neq t'}{\langle \mathbb{P}_t[\mathtt{fork}\ M] \parallel P; \Sigma\rangle \to \langle \mathbb{P}_t[\mathtt{return}\ t'] \parallel (\!|M|\!)_{t'} \parallel P; \Sigma\rangle}\ (\textsc{ForkP})$$

$$\frac{t' \notin \mathsf{threads}(P) \quad t \neq t'}{\langle \mathbb{T}_{t,k}[\mathtt{fork}\ M] \parallel P; \Sigma\rangle \to \langle \mathbb{T}_{t,k}[\mathtt{return}\ t'] \parallel (\!|M \triangleright \mathtt{return}|\!)_{t',k} \parallel P; \Sigma\rangle}\ (\textsc{ForkT})$$

$$\mathsf{threads}(T_{t_1} \parallel \cdots \parallel T_{t_n}) \triangleq \{t_1, \ldots, t_n\}$$

$$\frac{r \notin \mathrm{dom}(\Sigma_\Theta) \cup \mathrm{dom}(\Sigma_\Delta) \quad \Sigma'_\Theta = \Sigma_\Theta[r \mapsto M]}{\langle \mathbb{P}_t[\mathtt{newVar}\ M] \parallel P; \Sigma\rangle \to \langle \mathbb{P}_t[\mathtt{return}\ r] \parallel P; \Sigma'\rangle}\ (\textsc{NewP})$$

$$\frac{r \notin \mathrm{dom}(\Sigma_\Theta) \cup \mathrm{dom}(\Sigma_\Delta) \quad \Sigma'_\Delta = \Sigma_\Delta[r \mapsto (M,k)]}{\langle \mathbb{T}_{t,k}[\mathtt{newVar}\ M] \parallel P; \Sigma\rangle \to \langle \mathbb{T}_{t,k}[\mathtt{return}\ r] \parallel P; \Sigma'\rangle}\ (\textsc{NewT})$$

$$\frac{r \notin \mathrm{dom}(\Sigma_\Delta) \quad \Sigma_\Theta(r) = M}{\langle \mathbb{P}_t[\mathtt{readVar}\ r] \parallel P; \Sigma\rangle \to \langle \mathbb{P}_t[\mathtt{return}\ M] \parallel P; \Sigma\rangle}\ (\textsc{ReadP})$$

$$\frac{r \notin \mathrm{dom}(\Sigma_\Delta) \quad \Sigma_\Theta(r) = M \quad \Sigma'_\Delta = \Sigma_\Delta[r \mapsto (M,k)]}{\langle \mathbb{T}_{t,k}[\mathtt{readVar}\ r] \parallel P; \Sigma\rangle \to \langle \mathbb{T}_{t,k}[\mathtt{return}\ M] \parallel P; \Sigma'\rangle}\ (\textsc{ReadT})$$

$$\frac{M = \mathbb{E}[\mathtt{readVar}\ r] \quad \Sigma_\Delta(r) = (M', k)}{\langle (\!|M|\!)_t \parallel P; \Sigma\rangle \to \langle (\!|\mathbb{E}[\mathtt{return}\ M'] \triangleright \lambda\_.M|\!)_{t,k} \parallel P; \Sigma\rangle}\ (\textsc{ReadJoin})$$

$$\frac{\Sigma_\Delta(r) = (M,j) \quad \Sigma' = \Sigma[k \mapsto j]}{\langle \mathbb{T}_{t,k}[\mathtt{readVar}\ r] \parallel P; \Sigma\rangle \to \langle \mathbb{T}_{t,j}[\mathtt{return}\ M] \parallel P[k \mapsto j]; \Sigma'\rangle}\ (\textsc{ReadMerge})$$

$$\frac{r \notin \mathrm{dom}(\Sigma_\Delta) \quad \Sigma_\Theta(r) = N \quad \Sigma'_\Theta = \Sigma_\Theta[r \mapsto M]}{\langle \mathbb{P}_t[\mathtt{writeVar}\ r\ M] \parallel P; \Sigma\rangle \to \langle \mathbb{P}_t[\mathtt{return}\ ()] \parallel P; \Sigma'\rangle}\ (\textsc{WriteP})$$

$$\frac{r \notin \mathrm{dom}(\Sigma_\Delta) \quad \Sigma_\Theta(r) = N \quad \Sigma'_\Delta = \Sigma_\Delta[r \mapsto (M,k)]}{\langle \mathbb{T}_{t,k}[\mathtt{writeVar}\ r\ M] \parallel P; \Sigma\rangle \to \langle \mathbb{T}_{t,k}[\mathtt{return}\ ()] \parallel P; \Sigma'\rangle}\ (\textsc{WriteT})$$

$$\frac{M = \mathbb{E}[\mathtt{writeVar}\ r\ M'] \quad \Sigma_\Delta(r) = (M'', k) \quad \Sigma'_\Delta = \Sigma_\Delta[r \mapsto (M', k)]}{\langle (\!|M|\!)_t \parallel P; \Sigma\rangle \to \langle (\!|\mathbb{E}[\mathtt{return}\ ()] \triangleright \lambda\_.M|\!)_{t,k} \parallel P; \Sigma'\rangle}\ (\textsc{WriteJoin})$$

$$\frac{\Sigma_\Delta(r) = (N,j) \quad \Sigma' = \Sigma[k \mapsto j] \quad \Sigma'_\Delta = \Sigma_\Delta[k \mapsto (M,j)]}{\langle \mathbb{T}_{t,k}[\mathtt{writeVar}\ r\ M] \parallel P; \Sigma\rangle \to \langle \mathbb{T}_{t,j}[\mathtt{return}\ ()] \parallel P[k \mapsto j]; \Sigma'\rangle}\ (\textsc{WriteMerge})$$

Fig. 4. *OCTM* semantics: rules for $\to$.

Derivation rules in Figure 4 characterize the execution of pure (effect-free) terms, forks and memory operations both inside, and outside of some transaction; Derivation rules in Figure 5 characterize auxiliary operations for transaction management (e.g. creation) and their coordination (e.g distributed commits). Note that there are no derivation rules for `retry`. In fact, the meaning of `retry` is to inform the machine that choices made by the scheduler led to a state from which the program cannot proceed. From an implementation perspective this translates in the transaction being re-executed from the beginning (or a suitable check-point) following a different scheduling of its operations.

Due to lack of space we shall describe only a representative subset of the derivation rules from Figure 4 and Figure 5.

Reading a location falls into four cases depending on the location being claimed (i.e. occurring in Δ) and the reader being part of a transaction. The

$$\frac{k \notin \mathsf{transactions}(P)}{\langle (\!|\mathtt{atomic}\ M\ N \mathtt{>>=}\ N'|\!)_t \parallel P; \Sigma\rangle \xrightarrow{new_k} \langle (\!|M \triangleright N; N'|\!)_{t,k} \parallel P; \Sigma\rangle}\ (\textsc{Atomic})$$

$$\frac{\langle (\!|M|\!)_t; \Sigma\rangle \to^* \langle (\!|\mathtt{return}\ N|\!)_t; \Sigma'\rangle}{\langle \mathbb{P}_t[\mathtt{isolated}\ M]; \Sigma\rangle \to \langle \mathbb{P}_t[\mathtt{return}\ N]; \Sigma'\rangle}\ (\textsc{IsolatedP})$$

$$\frac{\mathtt{op} \in \{\mathtt{abort}, \mathtt{return}\} \quad \langle (\!|M \triangleright \mathtt{return}|\!)_{t,k}; \Sigma\rangle \to^* \langle (\!|\mathtt{op}\ N \triangleright \mathtt{return}|\!)_{t,k}; \Sigma'\rangle}{\langle \mathbb{T}_{t,k}[\mathtt{isolated}\ M]; \Sigma\rangle \to \langle \mathbb{T}_{t,k}[\mathtt{op}\ N]; \Sigma'\rangle}\ (\textsc{IsolatedT})$$

$$\frac{\Sigma'_\Delta = \mathsf{clean}(k, \Sigma_\Delta)}{\langle (\!|\mathtt{abort}\ M \triangleright N; N'|\!)_{t,k}; \Sigma\rangle \xrightarrow{ab_k M} \langle (\!|N(M) \mathtt{>>=}\ N'|\!)_t; \Sigma'\rangle}\ (\textsc{RaiseAbort1})$$

$$\frac{\Sigma'_\Delta = \mathsf{clean}(k, \Sigma_\Delta)}{\langle (\!|\mathtt{abort}\ M \triangleright N|\!)_{t,k}; \Sigma\rangle \xrightarrow{ab_k M} \langle (\!|N(M)|\!)_t; \Sigma'\rangle}\ (\textsc{RaiseAbort2})$$

$$\frac{\Sigma'_\Delta = \mathsf{clean}(k, \Sigma_\Delta)}{\langle (\!|M \triangleright N; N'|\!)_{t,k}; \Sigma\rangle \xrightarrow{\widehat{ab}_k M} \langle (\!|N(M) \mathtt{>>=}\ N'|\!)_t; \Sigma'\rangle}\ (\textsc{SigAbort1})$$

$$\frac{\Sigma'_\Delta = \mathsf{clean}(k, \Sigma_\Delta)}{\langle (\!|M \triangleright N|\!)_{t,k}; \Sigma\rangle \xrightarrow{\widehat{ab}_k M} \langle (\!|N(M)|\!)_t; \Sigma'\rangle}\ (\textsc{SigAbort2})$$

$$\frac{\langle P; \Sigma\rangle \xrightarrow{ab_k M} \langle P'; \Sigma'\rangle \quad \langle Q; \Sigma\rangle \xrightarrow{\widehat{ab}_k M} \langle Q'; \Sigma'\rangle}{\langle P \parallel Q; \Sigma\rangle \xrightarrow{ab_k M} \langle P' \parallel Q'; \Sigma'\rangle}\ (\textsc{AbBroadcast})$$

$$\frac{\Sigma'_\Theta = \mathsf{commit}(k, \Sigma_\Theta, \Sigma_\Delta) \quad \Sigma'_\Delta = \mathsf{clean}(k, \Sigma_\Delta)}{\langle (\!|\mathtt{return}\ M \triangleright N; N'|\!)_{t,k}; \Sigma\rangle \xrightarrow{co_k} \langle (\!|\mathtt{return}\ M \mathtt{>>=}\ N'|\!)_t; \Sigma'\rangle}\ (\textsc{Commit1})$$

$$\frac{\Sigma'_\Theta = \mathsf{commit}(k, \Sigma_\Theta, \Sigma_\Delta) \quad \Sigma'_\Delta = \mathsf{clean}(k, \Sigma_\Delta)}{\langle (\!|M \triangleright N|\!)_{t,k}; \Sigma\rangle \xrightarrow{co_k} \langle (\!|M|\!)_t; \Sigma'\rangle}\ (\textsc{Commit2})$$

$$\frac{\langle P; \Sigma\rangle \xrightarrow{co_k} \langle P'; \Sigma'\rangle \quad \langle Q; \Sigma\rangle \xrightarrow{co_k} \langle Q'; \Sigma'\rangle}{\langle P \parallel Q; \Sigma\rangle \xrightarrow{co_k} \langle P' \parallel Q'; \Sigma'\rangle}\ (\textsc{CoBroadcast})$$

$$\frac{\langle P; \Sigma\rangle \xrightarrow{\beta} \langle P'; \Sigma'\rangle \quad \mathsf{transactions}(\beta) \notin \mathsf{transactions}(Q)}{\langle P \parallel Q; \Sigma\rangle \xrightarrow{\beta} \langle P' \parallel Q; \Sigma\rangle}\ (\textsc{TrIgnore})$$

$$\mathsf{transactions}(T_{t_1} \parallel \cdots \parallel T_{t_n}) \triangleq \bigcup_{1 \le i \le n} \mathsf{transactions}(T_{t_i})$$
$$\mathsf{transactions}((\!|M|\!)_t) \triangleq \emptyset$$
$$\mathsf{transactions}((\!|M \triangleright M'; N|\!)_{t,k}) \triangleq \{k\}$$
$$\mathsf{transactions}((\!|M \triangleright N|\!)_{t,k}) \triangleq \{k\}$$

$$\mathsf{clean}(k, \Delta)(r) \triangleq \begin{cases} \bot & \text{if } \Delta(r) = (M, k) \\ \Delta(r) & \text{otherwise} \end{cases}$$

$$\mathsf{commit}(k, \Theta, \Delta)(r) \triangleq \begin{cases} M & \text{if } \Delta(r) = (M, k) \\ \Theta(r) & \text{otherwise} \end{cases}$$

Fig. 5. *OCTM* semantics: rules for $\xrightarrow{\beta}$.

rule (ReadP) characterize the reading of an unclaimed location from outside any transaction; the read is performed as expected leaving it unclaimed. Rule (ReadT) describes the reading of an unclaimed location r by a thread belonging to some transaction k; the side effect of the reading is r being claimed for k. Rules (ReadMerge) and (ReadJoin) cover the cases of readings against claimed locations. In the first scenario, the reading thread belongs to a transaction resulting in the two being merged, which is expressed by renaming its transaction via a substitution. In the remaining scenario, the reading thread does not belong to

any transaction and hence joins the transaction k which claimed the location. The newly created participant does not have any continuation since the whole term is set to be executed inside k; any other choice for splitting the term singling out a compensation would impose an artificial synchronization with the transaction commit. For a counter example, consider executing only the read operation inside the transaction and delaying everything after the commit; then concurrency will be clearly reduced. Because of the same reasoning, the whole term M is taken as the compensation of the participant.

Atomic transactions are created by the rule (Atomic); threads participating in this transaction are non-deterministically interleaved with other threads. The stronger requirement of isolation is offered by (IsolatedP) and (IsolatedT); note that their premises forbid thread or transaction creation.

Committing or aborting a transaction require a synchronization of its participants. In particular, an abort can be read as a participant vetoing the outcome of the transaction; this corresponds to (RaiseAbort1) and (RaiseAbort2). The information is then propagated by (AbBroadcast) and (TrIgnore) to any other participant to the transaction being aborted; these participants abort performing a transition described by either (SigAbort1) or (SigAbort2).

3 $TCCS^m$: CCS with open transactions

In order to assess the expressive power of $OCTM$, we have to compare it with a formal model for open transactions. To this end, in this section we recall $TCCS^m$ [10], a CCS-like calculus with open flat transactions: processes can synchronize even when belonging to different transactions, which in turn are joined into a distributed one. We refer to [10] for a detailed description of $TCCS^m$.

The syntax of $TCCS^m$ is defined by the following grammar

$$P ::= \textstyle\sum_{i=1}^{n} \alpha_i.P_i \mid \prod_{i=0}^{m} P_i \mid P\backslash L \mid X \mid \mu X.P \mid \langle P_1 \blacktriangleright P_2 \rangle \mid \langle\!\langle P_1 \triangleright_k P_2 \rangle\!\rangle \mid \mathsf{co}.P \quad (1)$$

where $\alpha_i ::= a \mid \bar{a} \mid \tau$, a ranges over a given set of visible actions A, L over subsets of A and the bijection $(\bar{\cdot}) : A \to A$ maps every action to its *coaction* as usual. The calculus extends CCS with three constructs which represent *inactive* transactions, *active* transactions and *commit* actions respectively. Transactions such as $\langle\!\langle P_1 \triangleright_k P_2 \rangle\!\rangle$ are formed by two processes with the former being executed atomically and the latter being executed whenever the transaction is aborted, i.e. as a *compensation*. Terms denoting active transactions expose also a name (k in the previous example) which is used to track transaction fusions. For instance, consider the process denoted by $\langle\!\langle P_1 \triangleright_j P_2 \rangle\!\rangle | \langle\!\langle Q_1 \triangleright_k Q_2 \rangle\!\rangle$ where P_1 and Q_1 synchronize on some $a \in A$; the result of this synchronization is the fusion of the transactions j and k i.e. $\langle\!\langle P_1' \triangleright_l P_2 \rangle\!\rangle | \langle\!\langle Q_1' \triangleright_l Q_2 \rangle\!\rangle$. The fusion makes explicit the dependency between j and k introduced by the synchronization and ties them to agree on commits. In this sense, P_1' and Q_1' are participants of a *distributed transaction* [6].

As in [10] we restrict ourselves to well-formed terms. Intuitively, a term is well-formed if active transactions occur only at the top-level and commit actions occur only in a transaction (active or inactive). To this end we introduce a *type system* for $TCCS^m$, whose rules are in Figure 6. Terms that cannot occur inside

$$\frac{\Gamma \vdash P : \mathsf{p}}{\Gamma \vdash P : \tau} \quad \frac{\Gamma \vdash P : \mathsf{p}}{\Gamma \vdash \mathsf{co}.P : \mathsf{c}} \quad \frac{\Gamma \vdash P : \tau}{\Gamma \vdash P \backslash L : \tau}$$

$$\frac{}{\Gamma \vdash X : \Gamma(X)} \quad \frac{\Gamma[X : \mathsf{p}] \vdash P : \mathsf{p}}{\Gamma \vdash \mu X.P : \mathsf{p}} \quad \frac{\Gamma[X : \mathsf{c}] \vdash P : \mathsf{c}}{\Gamma \vdash \mu X.P : \mathsf{c}} \quad \frac{\forall i\, \Gamma \vdash P_i : \tau}{\Gamma \vdash \prod P_i : \tau}$$

$$\frac{\forall i\, \Gamma \vdash P_i : \mathsf{p}}{\Gamma \vdash \sum \alpha_i.P_i : \mathsf{p}} \quad \frac{\forall i\, \Gamma \vdash \alpha_i.P_i : \mathsf{c}}{\Gamma \vdash \sum \alpha_i.P_i : \mathsf{c}} \quad \frac{\Gamma \vdash P : \mathsf{c} \quad \Gamma \vdash Q : \mathsf{p}}{\Gamma \vdash \langle\!\langle P \triangleright_k Q \rangle\!\rangle : \mathsf{t}} \quad \frac{\Gamma \vdash P : \mathsf{c} \quad \Gamma \vdash Q : \mathsf{p}}{\Gamma \vdash \langle P \blacktriangleright Q \rangle : \mathsf{p}}$$

Fig. 6. Simple types for $TCCS^m$.

a transaction have type t, terms that cannot occur outside a transaction have type c, and terms without such restrictions have type p; τ ranges over types.

Definition 1 (Well-formed $TCCS^m$ terms). *A $TCCS^m$ term P, described by the grammar in* (1)*, is said to be* well-formed *if, and only if, $\emptyset \vdash P : \mathsf{t}$. Well-formed terms form the set* Proc*.*

The operational semantics of well-formed $TCCS^m$ terms is given by the SOS in Figure 7 (further details can be found in [10]). The reduction semantics is given as a binary relation $\rightarrow$ defined by

$$P \rightarrow Q \iff^{\triangle} P \xrightarrow{\tau}_{\sigma} Q \vee P \xrightarrow{\beta} Q \vee P \xrightarrow{k(\tau)}_{\sigma} Q.$$

The first case is a synchronization between pure CCS processes. The second case corresponds to creation of new transactions and distributed commit or abort ($\beta \in \{\mathsf{new}k, \mathsf{co}k, \mathsf{ab}k\}$). The third case corresponds to synchronizations of processes inside a named (and possibly distributed) transaction. Notice that by (TSync) transaction fusion is *driven by communication* and that by (TSum) any pure CCS process can join and interact with a transaction.

4 Encoding $TCCS^m$ in $OCTM$

In this section, we prove that $OCTM$ is expressive enough to cover open transactions *a la* $TCCS^m$. To this end, we provide an encoding of $TCCS^m$ in $OCTM$. Basically, we have to implement transactions and CCS-like synchronizations using shared transactional variables and the `atomic` and `isolated` operators. The encoding is proved to be correct, in the sense that a $TCCS^m$ process presents a reduction if and only if also its encoding has the corresponding reduction.

Synchronization is implemented by means of shared transactional variables, one for each channel, that take values of type `ChState` (cf. Figure 9); this type has four constructors: one for each of the three messages of the communication protocol below plus a "nothing" one providing the default value. Let t_1 and t_2 be the identifiers of two threads simulating $a.P$ and $\overline{a}.Q$ respectively. The protocol is composed by the following four steps:

1. t_1 checks whether the channel is free and writes on the transactional variable modelling the channel a a nonce tagged with the constructor `M1`;
2. t_2 reads the variable for a and accepts the synchronization offered by the challenge `(M1 np)` adding a fresh nonce to it and writing back `(M2 np nq)`;

$$\frac{}{\sum \alpha_i.P_i \xrightarrow{\alpha_i}_\varepsilon P_i}\ (\textsc{Sum}) \qquad \frac{P \xrightarrow{a}_\varepsilon P' \quad Q \xrightarrow{\bar{a}}_\varepsilon Q'}{P|Q \xrightarrow{\tau}_\varepsilon P'|Q'}\ (\textsc{Sync}) \qquad \frac{}{\mu X.P \xrightarrow{\tau}_\varepsilon P[{}^{\mu X.P}/_X]}\ (\textsc{Rec})$$

$$\frac{P \xrightarrow{\alpha}_\sigma P' \quad \mathsf{img}(\sigma) \cap \mathsf{tn}(Q) = \emptyset}{P|Q \xrightarrow{\alpha}_\sigma P'|Q[\sigma]}\ (\textsc{ParL}) \qquad \frac{\tau \neq \alpha_j}{\sum \alpha_i.P_i \xrightarrow{k(\alpha_j)}_{\varepsilon \mapsto k} \langle\!\langle P_j | \mathsf{co} \triangleright_k \sum \alpha_i.P_i \rangle\!\rangle}\ (\textsc{TSum})$$

$$\frac{P \xrightarrow{\alpha}_\varepsilon P' \quad \tau \neq \alpha \quad l \neq k}{\langle\!\langle P \triangleright_l Q \rangle\!\rangle \xrightarrow{k(\alpha)}_{l \mapsto k} \langle\!\langle P' \triangleright_k Q \rangle\!\rangle}\ (\textsc{TAct}) \qquad \frac{P \xrightarrow{k(a)}_{i \mapsto k} P' \quad Q \xrightarrow{k(\bar{a})}_{j \mapsto k} Q'}{P|Q \xrightarrow{k(\tau)}_{i,j \mapsto k} P'[j \mapsto k]|Q'[i \mapsto k]}\ (\textsc{TSync})$$

$$\frac{P \xrightarrow{\alpha}_\sigma P' \quad \alpha \notin L}{P\backslash L \xrightarrow{\alpha}_\sigma P'\backslash L}\ (\textsc{Res}) \qquad \frac{P \xrightarrow{\tau}_\varepsilon P'}{\langle\!\langle P \triangleright_k Q \rangle\!\rangle \xrightarrow{\tau}_\varepsilon \langle\!\langle P' \triangleright_k Q \rangle\!\rangle}\ (\textsc{TTau}) \qquad \frac{}{\langle\!\langle P \triangleright_k Q \rangle\!\rangle \xrightarrow{\mathsf{ab}k} Q}\ (\textsc{TAb})$$

$$\frac{P \xrightarrow{\beta} P'}{P\backslash L \xrightarrow{\beta} P'\backslash L}\ (\textsc{TRes}) \qquad \frac{k \text{ fresh}}{\langle P \blacktriangleright Q \rangle \xrightarrow{\mathsf{new}k} \langle\!\langle P \triangleright_k Q \rangle\!\rangle}\ (\textsc{TNew}) \qquad \frac{\exists i\, P_i = \mathsf{co}.P_i'}{\langle\!\langle \prod P_i \triangleright_k Q \rangle\!\rangle \xrightarrow{\mathsf{co}k} \Psi_{id}(P)}\ (\textsc{TCo})$$

$$\frac{P \xrightarrow{\beta} P' \quad Q \xrightarrow{\beta} Q' \quad \beta \neq \mathsf{new}k}{P|Q \xrightarrow{\beta} P'|Q'}\ (\textsc{TB1}) \qquad \frac{P \xrightarrow{\beta} P' \quad \mathsf{tn}(\beta) \notin \mathsf{tn}(Q)}{P|Q \xrightarrow{\beta} P'|Q}\ (\textsc{TB2})$$

$$\Psi_\sigma(P) \triangleq \begin{cases} Q & \text{if } P = \mathsf{co}.Q \\ \Psi_\sigma(Q)\backslash L & \text{if } P = Q\backslash L \\ \sum \alpha_i.\Psi_\sigma(P_i) & \text{if } P = \sum \alpha_i.P_i \\ \prod \Psi_\sigma(P_i) & \text{if } P = \prod P_i \\ \mu X.\Psi_{\sigma[{}^{P}/_X]}(Q) & \text{if } P = \mu X.Q \\ P[\sigma] & \text{otherwise} \end{cases} \qquad \mathsf{tn}(P) \triangleq \begin{cases} \{k\} & \text{if } P = \langle\!\langle P \triangleright_k Q \rangle\!\rangle \\ \bigcup \mathsf{tn}(P_i) & \text{if } P = \prod P_i \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathsf{tn}(\beta) \triangleq \begin{cases} k & \text{if } \beta = \mathsf{new}k \\ k & \text{if } \beta = \mathsf{ab}k \\ k & \text{if } \beta = \mathsf{co}k \end{cases}$$

Fig. 7. $TCCS^m$ operational semantics.

3. t_1 reads the answer to its challenge and acknowledges the synchronization writing back the nonce it read tagged with the constructor `M3`;
4. t_2 reads the acknowledgement and frees the channel.

Each step has to be executed in isolation with respect to the interactions with the shared transactional variable a. Nonces are meant to correlate the steps only and hence can be easily implemented in *OCTM* by pairing thread identifiers with counter *a la* logical clock. If at any step a thread finds the channel in an unexpected state it means that the chosen scheduling has led to a state incoherent with respect to the above protocol; hence the thread executes a *retry*. This tells the scheduler to try another execution order; by fairness, we eventually find a scheduling such that the two processes do synchronize on a and these are the only executions leading to $P \mid Q$. The protocol is illustrated in Figure 8. If the synchronizing parties are involved in distinct transactions these are fused as a side effect of the interaction via the shared variable.

A choice like $\sum_{i=1}^{m} \alpha_i.P_i$ can be seen as a race of threads $t_1, \ldots, t_m$, each simulating a branch, to acquire a boolean transactional variable l (private to the group). Each t_i proceeds as follows. First, it checks l and if it is set, it returns void and terminates (another thread has already acquired it); otherwise it tries to set it while carrying out α_i, i.e. right before executing its last step of the

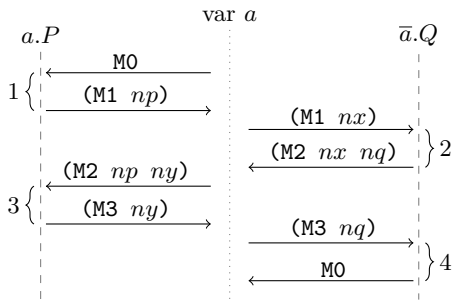

Fig. 8. Implementing $TCCS^m$ synchronization.

communication protocol. If the variable is acquired by another thread while t_i is finalizing α_i then t_i issues a `retry` to retract any effect of α_i. The $OCTM$ code implementing this protocol is shown in Figure 9.

Encoding of $TCCS^m$ We can now define the encoding $\eta : \mathsf{Proc} \to \mathsf{State}$, mapping well-formed $TCCS^m$ terms to states of the $OCTM$ abstract machine. Intuitively, a process $P \equiv \prod_{i=1}^{m} P_i$ is mapped into a state with a thread for each P_i and a variable for each channel in P. Clearly, a state of this form can be easily generated by a single $OCTM$ term which allocates all variables and forks the m threads. We have preferred to map $TCCS^m$ terms to $OCTM$ states for sake of simplicity.

The map η is defined by recursion along the derivation of $\emptyset \vdash P : \mathtt{t}$ and the number m of parallel components in $P \equiv \prod_{i=1}^{m} P_i$. This is handled by the auxiliary encoding $\varsigma : \mathsf{Proc} \times \mathsf{Heap} \to \mathsf{State}$ (up to choice of fresh names) whose second argument is used to track memory allocations. The base case is given by $m = 0$ and yields a state with no threads i.e. $\langle \mathbf{0}, \Theta, \varnothing \rangle$. The recursive step is divided in three subcases depending on the structure and type of P_1 ($m > 0$).

1. If $\emptyset \vdash P_1 : \mathsf{c}$ without top-level restrictions (i.e. $P_1 \not\equiv Q \setminus \{a_1, \ldots, a_{n+1}\}$) then

$$\varsigma(\textstyle\prod_{i=1}^{m+1} P_i, \Theta) \triangleq \langle (\!|\varrho(P_1)|\!)_{t_1} \parallel S; \Sigma \rangle$$

   where $\langle S; \Sigma \rangle = \varsigma(\prod_{j=1}^{m-1} P_{j+1}, \Theta)$ is the translation of the rest of P and t_1 is unique w.r.t. S (i.e. $t_1 \notin \mathsf{threads}(S)$). By hypothesis P_1 does not contain any top-level active transaction or parallel composition and hence can be translated directly into a $OCTM$-term by means of the encoding ϱ (cf. Figure 10) – $\varrho(P)$ contain a free variable for each unrestricted channel occurring in P.
2. If P_1 has a top-level restriction (i.e. $P_1 \equiv Q \setminus \{a_1, \ldots, a_{n+1}\}$) then

$$\varsigma(\textstyle\prod_{i=1}^{m+1} P_i, \Theta) \triangleq \langle S_1[r_1/a_1, \ldots r_{n+1}/a_{n+1}] \parallel S_2; \Theta_2[r_1, \ldots, r_{n+1} \mapsto \mathtt{M0}], \varnothing \rangle$$

   where $\langle S_1; \Theta_1, \varnothing \rangle = \varsigma(Q, \Theta)$ is the translation of the unrestricted process Q, $\langle S_2; \Theta_2, \varnothing \rangle = \varsigma(\prod_{j=1}^{m-1} P_{j+1}, \Theta_1)$ is the translation of the rest of P, all threads have a unique identifier $\mathsf{threads}(S_1) \cap \mathsf{threads}(S_2) = \emptyset$, the heap is extended with n channel variables fresh ($r_1, \ldots, r_{n+1} \notin \mathrm{dom}(\Theta_2)$) and known only to the translation of Q.

```
data Channel = OTVar ChState
data ChState = M1 Nonce | M2 Nonce Nonce | M3 Nonce | M0

tau l P = isolated do
  case (readVar l) of
    False → return ()
    True → chooseThis l >> P

chooseThis l = writeVar l False

eqOrRetry x y
  | x == y = return ()
  | otherwise = retry

bang x = fork x >> bang x

recv c l P = do
  nq ← newNonce
  isolated do
    case (readVar l) of
      False → return ()
      True → do
        chooseThis l
        case (readVar c) of
          (M1 nx) → writeVar c (M2 nx nq)
          _ → retry
  isolated do
    case (readVar c) of
      (M3 ny) → eqOrRetry ny nq >> writeVar c M0 >> P
      _ → retry

send c l P = do
  np ← newNonce
  isolated do
    case (readVar l) of
      False → return ()
      True → do
        chooseThis l
        case (readVar c) of
          M0 → writeVar c (M1 np)
          _ → retry
  isolated do
    case (readVar c) of
      (M2 nx ny) → eqOrRetry nx np >> writeVar c (M3 ny) >> P
      _ → retry
```

Fig. 9. Encoding channels and communication

$$
\begin{array}{rl}
\varrho(\sum_{i=1}^{m}\alpha_i P_i) \triangleq & \texttt{do} \\
& \quad l \leftarrow \texttt{newVar True} \\
& \quad \forall i \in \{1,\ldots,m\} \\
& \quad\quad \texttt{fork}\ \xi(\alpha_i, \texttt{l}, P_i) \\
\varrho(\prod_{i=0}^{m} P_i) \triangleq & \texttt{do} \\
& \quad \forall i \in \{0,\ldots,m\} \\
& \quad\quad \texttt{fork}\ \varrho(P_i) \\
\varrho(P \setminus L) \triangleq & \texttt{do} \\
& \quad \forall c \in L \\
& \quad\quad c \leftarrow \texttt{newVar M0} \\
& \quad \varrho(P) \\
\varrho(X) \triangleq & X \\
\varrho(P) & \\
\varrho(\texttt{co}.P) \triangleq & \texttt{do} \\
& \quad l \leftarrow \texttt{newVar True} \\
& \quad \texttt{send}\ co\ l\ \varrho(P)
\end{array}
\qquad
\begin{array}{rl}
\varrho(\mu X.P) \triangleq & \texttt{let}\ X = \varrho(P)\ \texttt{in} \\
\varrho(\langle P \blacktriangleright Q \rangle) \triangleq & \texttt{do} \\
& \quad co \leftarrow \texttt{newVar M0} \\
& \quad \texttt{atomic p}\ \varrho(Q) \\
& \quad \texttt{bang psi} \\
& \texttt{where} \\
& \quad \texttt{p} = \texttt{do} \\
& \quad\quad \varrho(P) \\
& \quad\quad \texttt{fork (abort ())} \\
& \quad\quad \texttt{psi} \\
& \quad \texttt{psi} = \texttt{do} \\
& \quad\quad l \leftarrow \texttt{newVar True} \\
& \quad\quad \texttt{recv}\ co\ l\ \texttt{return} \\
\xi(\alpha_i, l, P_i) \triangleq & \begin{cases} \texttt{recv}\ \alpha_i\ l\ \varrho(P_i) & \text{if } \alpha_i = c \\ \texttt{send}\ \overline{\alpha_i}\ l\ \varrho(P_i) & \text{if } \alpha_i = \overline{c} \\ \texttt{tau}\ l\ \varrho(P_i) & \text{if } \alpha_i = \tau \end{cases}
\end{array}
$$

Fig. 10. Encoding $TCCS^m$ terms of type c

3. If $P_1 \equiv \langle\!\langle Q_1 \triangleright_k Q_2 \rangle\!\rangle$ is an active transaction then

$$\varsigma(\prod_{i=1}^{m+1} P_i, \Theta) \triangleq \langle S_{co} \parallel S_{ab} \parallel S_1[r_{co}/co] \parallel S_2; \Theta_2[r_l \mapsto True, r_{co} \mapsto \texttt{M0}], \varnothing \rangle$$
$$S_{co} = (\!|\texttt{recv}\ r_l\ r_{co} \triangleright \varrho(Q_1); \texttt{bang (recv (newVar True)}\ r_{co}\texttt{)}|\!)_{t_{co},k}$$
$$S_{ab} = (\!|\texttt{abort ()} \triangleright \texttt{return}|\!)_{t_{ab},k}$$

where $\langle S_1; \Theta_1, \varnothing \rangle = \varsigma(Q_1, \Theta)$, $\langle S_2; \Theta_2, \varnothing \rangle = \varsigma(\prod_{j=1}^{m-1} P_{j+1}, \Theta_2)$ (like above), the thread S_{ab} is always ready to abort k as in (TAB) and S_{co} awaits on the private channel r_{co} a thread from S_1 to reach a commit and, after its commit, collects all remaining synchronizations on r_{co} to emulate the effect of Ψ (cf. Figure 7). Finally, all threads have to be uniquely identified: $\mathsf{threads}(S_1) \cap \mathsf{threads}(S_2) = \emptyset$ and $t_{co}, t_{ab} \notin \mathsf{threads}(S_1) \cup \mathsf{threads}(S_2)$

Remark 1. The third case of the definition above can be made more precise (at the cost of a longer definition) since the number of commits to be collected can be inferred from Q mimicking the definition of Ψ. This solution reduces the presence of dangling auxiliary processes and transaction fusions introduced by the cleaning process.

Like ϱ, $\varsigma(P, \Theta)$ contains a free variable for each unrestricted channel in P. Finally, the encoding η is defined on each $P \in \mathsf{Proc}$ as:

$$\eta(P) \triangleq \langle S[r_1/a_1, \ldots r_n/a_n]; \Theta[r_1, \ldots, r_n \mapsto \texttt{M0}], \varnothing \rangle$$

where $\langle S; \Theta, \varnothing \rangle = \varsigma(P, \varnothing)$, $\{a_1, \ldots, a_n\} \subseteq A$ is the set of channels occurring in P, and $\{r_1, \ldots, r_n\} \subseteq \mathsf{Loc}$.

Adequacy of translation In order to prove that the translation η preserves the behaviour of $TCCS^m$ processes, we construct a *simulation relation* $\mathcal{S}$ between well-formed $TCCS^m$ processes and states of $OCTM$. The basic idea is that a single step of P is simulated by a sequence of reductions of $\eta(P)$, and $\eta(P)$ does not exhibit behaviours which are not exhibited by P. To this end we define an appropriate notion of *star simulation*, akin to [11]:

Definition 2 (Star simulation). *A relation $\mathcal{S} \subseteq \mathsf{Proc} \times \mathsf{State}$ is a* star simulation *if for all $(P, \langle S; \Sigma\rangle) \in \mathcal{S}$:*

1. *for all Q such that $P \xrightarrow{\tau}_\sigma Q$ or $P \xrightarrow{k(\tau)}_\sigma Q$, there exist S', Σ' such that $\langle S; \Sigma\rangle \to^* \langle S'; \Sigma'\rangle$ and $(Q, \langle S'; \Sigma'\rangle) \in \mathcal{S}$;*
2. *for all Q such that $P \xrightarrow{\beta} Q$, there exist S', Σ' such that $\langle S; \Sigma\rangle \xrightarrow{\beta}{}^* \langle S'; \Sigma'\rangle$ and $(Q, \langle S'; \Sigma'\rangle) \in \mathcal{S}$.*
3. *for all S', Σ' such that $\langle S; \Sigma\rangle \to \langle S'; \Sigma'\rangle$, there exist Q, S'', Σ'' such that $(Q, \langle S''; \Sigma''\rangle) \in \mathcal{S}$ and one of the following holds:*
   - *$P \xrightarrow{\tau}_\sigma Q$ or $P \xrightarrow{k(\tau)}_\sigma Q$, and $\langle S'; \Sigma'\rangle \to^* \langle S''; \Sigma''\rangle$*
   - *$P \xrightarrow{\beta}_\epsilon Q$ and $\langle S'; \Sigma'\rangle \xrightarrow{\beta}{}^* \langle S''; \Sigma''\rangle$.*

where β-labels of the two transition relations are considered equivalent whenever are both commits or both aborts for the same transaction name. We say that P is star-simulated by *$\langle S; \Sigma\rangle$ if there exists a star-simulation $\mathcal{S}$ such that $(P, \langle S; \Sigma\rangle) \in \mathcal{S}$. We denote by $\overset{*}{\approx}$ the largest star simulation.*

Another technical issue is that two equivalent $TCCS^m$ processes can be translated to $OCTM$ states which differ only on non-observable aspects, like name renamings, terminated threads, etc. To this end, we need to consider $OCTM$ states up-to an equivalence relation $\cong_t \subseteq \mathsf{State} \times \mathsf{State}$, which we define next.

Definition 3. *Two OCTM states are* transaction-equivalent*, written $\langle S_1; \Sigma_1\rangle \cong_t \langle S_2; \Sigma_2\rangle$, when they are equal up to:*

- *renaming of transaction and thread names;*
- terminated *threads, i.e. threads of one of the following forms: $(\!|\texttt{return}\ M|\!)_t$, $(\!|\texttt{abort}\ M|\!)_t$, $(\!|\texttt{return} \triangleright \texttt{return}|\!)_{t,k}$, $(\!|\texttt{abort} \triangleright \texttt{return}|\!)_{t,k}$, $(\!|\texttt{psi}|\!)_t$;*
- *threads blocked in synchronizations on co variables.*

Definition 4. *Let $P \in Proc$ be a well-formed process and $\langle S; \Sigma\rangle$ be a state. P is* star simulated by *$\langle S; \Sigma\rangle$* up to *$\cong_t$ if $(P, \langle S; \Sigma\rangle) \in \overset{*}{\approx} \circ \cong_t$.*

We are now ready to state our main adequacy result, which is a direct consequence of the two next technical lemmata.

Lemma 1. *For all $P, Q \in Proc$ the following hold true:*

1. *if $P \xrightarrow{\tau}_\sigma Q$ or $P \xrightarrow{k(\tau)}_\sigma Q$, there exist S, Σ such that $\eta(P) \to^* \langle S; \Sigma\rangle$ and $\langle S; \Sigma\rangle \cong_t \eta(Q)$;*
2. *if $P \xrightarrow{\beta} Q$, there exist S, Σ such that $\eta(P) \xrightarrow{\beta}{}^* \langle S; \Sigma\rangle$ and $\langle S; \Sigma\rangle \cong_t \eta(Q)$.*

Proof. By induction on the syntax of P; see [15]. □

Lemma 2. *For $P \in Proc$, for all S, Σ, if $\eta(P) \to \langle S; \Sigma \rangle$ then there exist Q, S', Σ' such that $\langle S'; \Sigma' \rangle \cong_t \eta(Q)$ and one of the following holds:*

- $P \xrightarrow{\tau}_\sigma Q$ *or* $P \xrightarrow{k(\tau)}_\sigma Q$, *and* $\langle S; \Sigma \rangle \to^* \langle S'; \Sigma' \rangle$;
- $P \xrightarrow{\beta}_\epsilon Q$ *and* $\langle S; \Sigma \rangle \xrightarrow{\beta}^* \langle S'; \Sigma' \rangle$.

Proof. By induction on the semantics of $\eta(P)$; see [15]. □

Theorem 1. *For all $P \in Proc$, P is star simulated by $\eta(P)$ up to $\cong_t$.*

5 Conclusions and future work

In this paper we have introduced *OCTM*, a higher-order language extending the concurrency model of STM Haskell with composable *open (multi-thread)* transactions. In this language, processes can *join* transactions and transactions can *merge* at runtime. These interactions are driven only by access to shared transactional memory, and hence are implicit and loosely coupled. To this end, we have separated the isolation aspect from atomicity: the `atomic` construct ensures "all-or-nothing" execution but not isolation, while the new constructor `isolated` can be used to guarantee isolation when needed. In order to show the expressive power of *OCTM*, we have provided an adequate implementation in it of $TCCS^m$, a recently introduced model of open transactions with CCS-like communication. As a side result, we have given a simple typing system for capturing $TCCS^m$ well-formed terms.

Several directions for future work stem from the present paper. First, we plan to implement *OCTM* along the line of STM Haskell, but clearly the basic ideas of *OCTM* are quite general and can be applied to other STM implementations, like C/C++ LibCMT and Java Multiverse. Then, we can use $TCCS^m$ as an *exogenous orchestration language* for *OCTM*: the *behaviour* of a transactional distributed system can be described as a $TCCS^m$ term, which can be translated into a *skeleton* in *OCTM* using the encoding provided in this paper; then, the programmer has only to "fill in the gaps". Thus, $TCCS^m$ can be seen as a kind of "global behavioural type" for *OCTM*.

In fact, defining a proper behavioural typing system for transactional languages like *OCTM* is another interesting future work. Some preliminary experiments have shown that $TCCS^m$ is not enough expressive for modelling the dynamic creation of resources (locations, threads, etc.). We think that a good candidate could be a variant of $TCCS^m$ with local names and scope extrusions, i.e., a "transactional π-calculus".

Being based on CCS, communication in $TCCS^m$ is synchronous; however, nowadays asynchronous models play an important rôle (see e.g. actors, event-driven programming, etc.). It may be interesting to generalize the discussion so as to consider also this case, e.g. by defining an actor-based calculus with open transactions. Such a calculus can be quite useful also for modelling speculative reasoning for cooperating systems [13, 14]. A local version of actor-based open transactions can be implemented in *OCTM* using lock-free data structures (e.g., message queues) in shared transactional memory.

Acknowledgements We thank the anonymous referees for useful remarks and suggestions on the preliminary version of this paper. This work is partially supported by MIUR PRIN project 2010LHT4KM, *CINA*.

References

1. M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. *ACM Trans. Program. Lang. Syst.*, 33(1):2, 2011.
2. R. Bruni, H. C. Melgratti, and U. Montanari. Nested commits for mobile calculi: Extending join. In J. Lévy, E. W. Mayr, and J. C. Mitchell, editors, *Proc. TCS*, volume 155 of *IFIP*, pages 563–576. Kluwer/Springer, 2004.
3. V. Danos and J. Krivine. Transactions in RCCS. In M. Abadi and L. de Alfaro, editors, *Proc. CONCUR*, volume 3653 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2005.
4. E. de Vries, V. Koutavas, and M. Hennessy. Communicating transactions (extended abstract). In P. Gastin and F. Laroussinie, editors, *Proc. CONCUR*, volume 6269 of *Lecture Notes in Computer Science*, pages 569–583. Springer, 2010.
5. E. de Vries, V. Koutavas, and M. Hennessy. Liveness of communicating transactions (extended abstract). In K. Ueda, editor, *Proc. APLAS*, volume 6461 of *Lecture Notes in Computer Science*, pages 392–407. Springer, 2010.
6. J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions Database Systems*, 31(1):133–160, 2006.
7. T. Harris, S. Marlow, S. L. P. Jones, and M. Herlihy. Composable memory transactions. In *Proc. PPOPP*, pages 48–60, 2005.
8. 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*, pages 289–300. ACM, 1993.
9. M. Herlihy and N. Shavit. Transactional memory: beyond the first two decades. *SIGACT News*, 43(4):101–103, 2012.
10. V. Koutavas, C. Spaccasassi, and M. Hennessy. Bisimulations for communicating transactions - (extended abstract). In A. Muscholl, editor, *Proc. FOSSACS*, volume 8412 of *Lecture Notes in Computer Science*, pages 320–334. Springer, 2014.
11. X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.
12. M. Lesani and J. Palsberg. Communicating memory transactions. In C. Cascaval and P. Yew, editors, *Proc. PPOPP*, pages 157–168. ACM, 2011.
13. 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, *Proc. CLMAS*, volume 6245 of *Lecture Notes in Computer Science*, pages 49–64. Springer, 2010.
14. 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.
15. M. Miculan, M. Peressotti, and A. Toneguzzo. Open transactions on shared memory. *CoRR*, abs/1503.09097, 2015.
16. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.
17. N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.