A Concurrent Calculus with Atomic Transactions*

Lucia Acciai[1], Michele Boreale[2], and Silvano Dal Zilio[1]

[1] LIF, CNRS and Université de Provence, France
[2] Dipartimento di Sistemi e Informatica, Università di Firenze, Italy

Abstract. The *Software Transactional Memory* (STM) model is an original approach for controlling concurrent accesses to resources without the need for explicit lock-based synchronization mechanisms. A key feature of STM is to provide a way to group sequences of read and write actions inside *atomic blocks*, similar to database transactions, whose whole effect should occur atomically.

In this paper, we investigate STM from a process algebra perspective and define an extension of asynchronous CCS with atomic blocks of actions. We show that the addition of atomic transactions results in a very expressive calculus, enough to easily encode other concurrent primitives such as guarded choice and multiset-synchronization (à la join-calculus). The correctness of our encodings is proved using a suitable notion of bisimulation equivalence. The equivalence is then applied to prove interesting "laws of transactions" and to obtain a simple normal form for transactions.

1 Introduction

The craft of programming concurrent applications is about mastering the strains between two key factors: getting hold of results as quickly as possible, while ensuring that only correct results (and behaviors) are observed. To this end, it is vital to avoid unwarranted access to shared resources. The *Software Transactional Memory* (STM) [18] model is an original approach for controlling concurrent accesses to resources without using explicit lock-based synchronization mechanisms. Similarly to database transactions, the STM approach provides a way to group sequences of read and write actions inside *atomic blocks* whose whole effect should occur atomically. The STM model has several advantages. Most notably, it dispenses the programmer from the need to explicitly manipulate locks, a task widely recognized as difficult and error-prone. Moreover, atomic transactions provide a clean conceptual basis for concurrency control, which should ease the verification of concurrent programs. Finally, the model is effective: there exist several STM implementations for designing software for multiprocessor systems; these applications exhibit good performances in practice (compared to equivalent, hand-crafted, code using locks).

We investigate the STM model from a process algebra perspective and define an extension of asynchronous CCS [20] with atomic blocks of actions. We call this calculus ATCCS. The choice of a dialect of CCS is motivated by an attention to economy: to focus on STM primitives, we study a calculus as simple as possible and dispense with

* This work was partially supported by the French ANR ARASSIA project COPS and the EU FET-GC2 initiative, project SENSORIA.

orthogonal issues such as values, mobility of names or processes, *etc*. We believe that our work could be easily transferred to a richer setting. Our goal is not only to set a formal ground for reasoning on STM implementations but also to understand how this model fits with other concurrency control mechanisms. We also view this calculus as a test bed for extending process calculi with atomic transactions. This is an interesting direction for investigation since, for the most part, works that mix transactions with process calculi consider *compensating transactions*, see e.g. [3,5,7,8,9,11,19].

The idea of providing hardware support for software transactions originated from works by Herlihy and Moss [18] and was later extended by Shavit and Touitou [22] to software-only transactional memory. Transactions are used to protect the execution of an atomic block. Intuitively, each thread that enters a transaction takes a snapshot of the shared memory (the global state). The evaluation is optimistic and all actions are performed on a copy of the memory (the local state). When the transaction ends, the snapshot is compared with the current state of the memory. There are two possible outcomes: if the check indicates that concurrent writes have occurred, the transaction aborts and is rescheduled; otherwise, the transaction is committed and its effects are propagated instantaneously. Very recently, Harris et al. [17] have proposed a (combinator style) language of transactions that enables arbitrary atomic operations to be composed into larger *atomic expressions*. We base the syntax of ATCCS on the operators defined in [17].

The main contributions of this work are: (1) the definition of a process calculus with atomic transactions; and (2) the definition of an asynchronous bisimulation equivalence $\approx_{\mathsf{a}}$ that allows compositional reasoning on transactions. We also have a number of more specific technical results. We show that ATCCS is expressive enough to easily encode interesting concurrent primitives, such as (preemptive versions of) guarded choice and multiset-synchronization, and the leader election problem (Section 3). Next, we define an equivalence between atomic expressions $\backsimeq$ and prove that $\approx_{\mathsf{a}}$ and $\backsimeq$ are congruences (Section 4). These equivalences are used to prove the correctness of our encodings, to prove interesting "behavioral laws of transactions" and to define a simple normal form for transactions. We also show that transactions (modulo $\backsimeq$) have an algebraic structure close to that of a bound semilattice, an observation that could help improve the design of the transaction language. The proofs of the main theorems can be found in a long version of this paper [1].

2 The Calculus

We define the syntax and operational semantics of ATCCS, which is essentially a cut down version of asynchronous CCS, without choice and relabeling operators, equipped with atomic blocks and constructs for composing (transactional) sequences of actions.

Syntax of Processes and Atomic Expressions. The syntax of ATCCS, given in Table 1, is divided into syntactical categories that define a stratification of terms. The definition of the calculus depends on a set of names, ranged over by $a, b, \ldots$. As in CCS, names model communication channels used in process synchronization, but they also occur as objects of read and write actions in atomic transactions.

Atomic expressions, ranged over by $M, N, \ldots$, are used to define sequences of actions whose effect should happen atomically. Actions $\mathtt{rd}\,a$ and $\mathtt{wt}\,a$ represent attempts to input and output to the channel a. Instead of using snapshots of the state for managing transaction, we use a log-based approach. During the evaluation of an atomic block, actions are recorded in a private log δ (a sequence $\alpha_1 \ldots \alpha_n$) and have no effects outside the scope of the transaction until it is committed. The action `retry` aborts an atomic expression unconditionally and starts its execution afresh, with an empty log ε. The termination action `end` signals that an expression is finished and should be committed. If the transaction can be committed, all actions in the log are performed at the same time and the transaction is closed, otherwise the transaction aborts. Finally, transactions can be composed using the operator `orElse`, which implements (preemptive) alternatives between expressions. In M `orElse` N, the expression N is executed if M aborts and has the behavior of M otherwise.

Processes, ranged over by $P, Q, R, \ldots$, model concurrent systems of communicating agents. We have the usual operators of CCS: the empty process, $\mathbf{0}$, the parallel composition $P \mid Q$, and the input prefix $a.P$. There are some differences though. The calculus is asynchronous, meaning that a process cannot block on output actions. Also, we use *replicated input* $*\, a\,.P$ instead of recursion (this does not change the expressiveness of the calculus) and we lack the choice and relabeling operators of CCS. The hiding operator $P \setminus^n a$ bounds the scope of name a to P (we consider processes up-to α-renaming of bound names; we discuss the meaning of the annotation n in page 53). Finally, the main addition is the presence of the operator $\mathtt{atom}(M)$, which models a transaction that safeguards the expression M. The process $\{\!|A|\!\}_M$ represents the ongoing evaluation of an atomic block M: the subscript is used to keep the initial code of the transaction, in case it is aborted and executed afresh, while A holds the remaining actions that should be performed.

An *ongoing atomic block*, $A, B, \ldots$, is essentially an atomic expression enriched with an *evaluation state* σ and a *log* δ of the currently recorded actions. A state σ is a multiset of names that represents the output actions visible to the transaction when it was initiated. (This notion of state bears some resemblance with tuples space in coordination calculi, such as Linda [10].) When a transaction ends, the state σ recorded in the block $(M)_{\sigma;\delta}$ (the state at the initiation of the transaction) can be compared with the current state (the state when the transaction ends) to check if other processes have concurrently made changes to the global state, in which case the transaction should be aborted.

Notation. In the following, we write $\sigma \uplus \{a\}$ for the multiset σ enriched with the name a and $\sigma \setminus \sigma'$ for the multiset obtained from σ by removing elements found in σ', that is the smallest multiset σ'' such that $\sigma \subseteq \sigma' \uplus \sigma''$. The symbol $\emptyset$ stands for the empty multiset while $\{a^n\}$ is the multiset composed of exactly n copies of a, where $\{a^0\} = \emptyset$.

Given a log δ, we use the notation $\mathrm{WT}(\delta)$ for the multiset of names which appear as objects of a write action in δ. Similarly, we use the notation $\mathrm{RD}(\delta)$ for the multiset of names that are objects of read actions. The functions WT and RD may be inductively defined as follows: $\mathrm{WT}(\mathtt{wt}\,a.\delta) = \mathrm{WT}(\delta) \uplus \{a\}$; $\mathrm{RD}(\mathtt{rd}\,a.\delta) = \mathrm{RD}(\delta) \uplus \{a\}$; $\mathrm{WT}(\mathtt{rd}\,a.\delta) = \mathrm{WT}(\delta)$; $\mathrm{RD}(\mathtt{wt}\,a.\delta) = \mathrm{RD}(\delta)$ and $\mathrm{WT}(\varepsilon) = \mathrm{RD}(\varepsilon) = \varepsilon$.

Table 1. Syntax of ATCCS: Processes and Atomic Expressions

Actions $\alpha, \beta ::=$	$\mathtt{rd}\, a$	(tentative) read access to a
	$\mid\ \mathtt{wt}\, a$	(tentative) write access to a
(Atomic) Expressions $M, N ::=$	$\mathtt{end}$	termination
	$\mid\ \mathtt{retry}$	abort and retry the current atomic block
	$\mid\ \alpha.M$	action prefix
	$\mid\ M\ \mathtt{orElse}\ N$	alternative
Ongoing expressions $A, B ::=$	$(M)_{\sigma;\delta}$	execution of M with state σ and log δ
	$\mid\ A\ \mathtt{orElse}\ B$	ongoing alternative
Processes $P, Q ::=$	$\mathbf{0}$	nil
	$\mid\ \overline{a}$	(asynchronous) output
	$\mid\ a.P$	input
	$\mid\ *a.P$	replicated input
	$\mid\ P \mid Q$	parallel composition
	$\mid\ P \backslash^n a$	hiding
	$\mid\ \mathtt{atom}(M)$	atomic block
	$\mid\ \{\!\mid A \mid\!\}_M$	ongoing atomic block

Example: Composing Synchronization. Before we describe the meaning of processes, we try to convey the semantics of ATCCS (and the usefulness of the atomic block operator) using a simple example. We take the example of a concurrent system with two memory cells, M_1 and M_2, used to store integers. We consider here a straightforward extension of the calculus with "value-passing." In this setting, we can model a cell with value v by an output $\overline{m_i}!v$ and model an update by a process of the form $m_i?x.(\overline{m_i}!v' \mid \dots)$. With this encoding, the channel name m_i acts as a lock protecting the shared resource M_i.

Assume now that the values of the cells should be synchronized to preserve a global invariant on the system. For instance, we model a flying aircraft, each cell store the pitch of an aileron and we need to ensure that the aileron stay aligned (that the values of the cells are equal). A process testing the validity of the invariant is for example P_1 below (we suppose that a message on the reserved channel err triggers an alarm). There are multiple design choices for resetting the value of both cells to 0, e.g. P_2 and P_3.

$$P_1 \triangleq m_1?x.m_2?y.\text{if}\ x\,!\!=y\ \text{then}\,\overline{err}!$$

$$P_2 \triangleq m_2?x.m_1?y.\big(\overline{m_1}!0 \mid \overline{m_2}!0\big) \qquad P_3 \triangleq m_1?x.\big(\overline{m_1}!0 \mid m_2?y.\overline{m_2}!0\big)$$

Table 2. Operational Semantics: Processes

$$(\text{OUT})\ \overline{a};\sigma \to \mathbf{0};\sigma \uplus \{a\} \qquad (\text{REP})\ *a.P;\sigma \uplus \{a\} \to P \mid *a.P;\sigma$$

$$(\text{IN})\ a.P;\sigma \uplus \{a\} \to P;\sigma \qquad (\text{COM})\ \frac{P;\sigma \to P';\sigma \uplus \{a\} \quad Q;\sigma \uplus \{a\} \to Q';\sigma}{P|Q \to P' \mid Q'}$$

$$(\text{PARL})\ \frac{P;\sigma \to P';\sigma'}{P \mid Q;\sigma \to P' \mid Q;\sigma'} \qquad (\text{HID})\ \frac{P;\sigma \uplus \{a^n\} \to P';\sigma' \uplus \{a^m\} \qquad a \notin \sigma,\sigma'}{P\backslash^n a;\sigma \to P'\backslash^m a;\sigma'}$$

$$(\text{PARR})\ \frac{Q;\sigma \to Q';\sigma'}{P \mid Q;\sigma \to P \mid Q';\sigma'} \qquad (\text{ATST})\ \mathtt{atom}(M);\sigma \to \{\!|(M)_{\sigma;\varepsilon}|\!\}_M;\sigma$$

$$(\text{ATPASS})\ \frac{A \to A'}{\{\!|A|\!\}_M;\sigma \to \{\!|A'|\!\}_M;\sigma} \qquad (\text{ATRE})\ \{\!|(\mathtt{retry})_{\sigma';\delta}|\!\}_M;\sigma \to \mathtt{atom}(M);\sigma$$

$$(\text{ATFAIL})\ \frac{\text{RD}(\delta) \not\subseteq \sigma}{\{\!|(\mathtt{end})_{\sigma';\delta}|\!\}_M;\sigma \to \mathtt{atom}(M);\sigma}$$

$$(\text{ATOK})\ \frac{\text{RD}(\delta) \subseteq \sigma \quad \sigma = \sigma'' \uplus \text{RD}(\delta) \quad \text{WT}(\delta) = \{a_1,\dots,a_n\}}{\{\!|(\mathtt{end})_{\sigma';\delta}|\!\}_M;\sigma \to \overline{a_1} \mid \cdots \mid \overline{a_n};\sigma''}$$

Each choice exemplifies a problem with lock-based programming. The composition of P_1 with P_2 leads to a race condition where P_1 acquire the lock on M_1, P_2 on M_2 and each process gets stuck. The composition of P_1 and P_3 may break the invariant (the value of M_1 is updated too quickly). A solution in the first case is to strengthen the invariant and enforce an order for acquiring locks, but this solution is not viable in general and opens the door to *priority inversion* problems. Another solution is to use an additional (master) lock to protect both cells, but this approach obfuscate the code and significantly decreases the concurrency of the system.

Overall, this simple example shows that synchronization constraints do not compose well when using locks. This situation is consistently observed (and bears a resemblance to the inheritance anomaly problem found in concurrent object-oriented languages). The approach advocated in this paper is to use atomic transactions. In our example, the problem is solved by simply wrapping the two operations in a transaction, like in the process $\mathtt{atom}\big(\mathtt{rd}\,(m_2?y).\mathtt{wt}\,(m_2!0).\mathtt{rd}\,(m_1?x).\mathtt{wt}\,(m_1!0)\big)$, which ensures that all cell updates are effected atomically. More examples may be found on the paper on composable memory transactions [17], which makes a compelling case that "even correctly-implemented concurrency abstractions cannot be composed together to form larger abstractions."

Operational Semantics. Like for the syntax, the semantics of ATCCS is stratified in two levels: there is one reduction relation for processes and a second for atomic expressions. With a slight abuse of notation, we use the same symbol ($\to$) for both relations.

Table 3. Operational Semantics: Ongoing Atomic Expression

$$(\textsc{ARdOk})\ \frac{\textsc{rd}(\delta)\uplus\{a\}\subseteq\sigma}{(\mathtt{rd}\,a.M)_{\sigma;\delta}\rightarrow(M)_{\sigma;\delta.\mathtt{rd}\,a}}\qquad(\textsc{ARdF})\ \frac{\textsc{rd}(\delta)\uplus\{a\}\not\subseteq\sigma}{(\mathtt{rd}\,a.M)_{\sigma;\delta}\rightarrow(\mathtt{retry})_{\sigma;\delta}}$$

$$(\textsc{AWr})\quad(\mathtt{wt}\,a.M)_{\sigma;\delta}\rightarrow(M)_{\sigma;\delta.\mathtt{wt}\,a}$$

$$(\textsc{AOI})\quad(M_1\ \mathtt{orElse}\ M_2)_{\sigma;\delta}\rightarrow(M_1)_{\sigma;\delta}\ \mathtt{orElse}\ (M_2)_{\sigma;\delta}$$

$$(\textsc{AOF})\ (\mathtt{retry})_{\sigma;\delta}\ \mathtt{orElse}\ B\rightarrow B\qquad(\textsc{AOE})\ (\mathtt{end})_{\sigma;\delta}\ \mathtt{orElse}\ B\rightarrow(\mathtt{end})_{\sigma;\delta}$$

$$(\textsc{AOL})\ \frac{A\rightarrow A'}{A\ \mathtt{orElse}\ B\rightarrow A'\ \mathtt{orElse}\ B}\qquad(\textsc{AOR})\ \frac{B\rightarrow B'}{A\ \mathtt{orElse}\ B\rightarrow A\ \mathtt{orElse}\ B'}$$

Reduction for Processes. Table 2 gives the semantics of processes. A reduction is of the form $P;\sigma\rightarrow P';\sigma'$ where σ is the state of P. The state σ records the names of all output actions visible to P when reduction happens. It grows when an output is reduced, (OUT), and shrinks in the case of inputs, (IN) and (REP). A parallel composition evolves if one of the component evolves or if both can synchronize, rules (PARL), (PARR) and (COM). In a hiding $P\backslash^n a$, the annotation n is an integer denoting the number of outputs on a that are visible to P. Intuitively, in a "configuration" $P\backslash^n a;\sigma$, the outputs visible to P are those in $\sigma\uplus\{a^n\}$. This extra annotation is necessary because the scope of a is restricted to P, hence it is not possible to have outputs on a in the global state. Rule (HID) allows synchronization on the name a to happen inside a hiding. For instance, we have $(P\mid\overline{a})\backslash^n a;\sigma\rightarrow P\backslash^{n+1}a;\sigma$.

The remaining reduction rules govern the evolution of atomic transactions. Like in the case of (COM), all those rules, but (ATOK), leave the global state unchanged. Rule (ATST) deals with the initiation of an atomic block $\mathtt{atom}(M)$: an ongoing block $\{\!|(M)_{\sigma;\varepsilon}|\!\}_M$ is created which holds the current evaluation state σ and an empty log ε. An atomic block $\{\!|A|\!\}_M$ reduces when its expression A reduces, rule (ATPASS). (The reduction relation for ongoing expressions is defined by the rules in Table 3.) Rules (ATRE), (ATFAIL) and (ATOK) deal with the completion of a transaction. After a finite number of transitions, the evaluation of an ongoing expression will necessarily result in a fail state, $(\mathtt{retry})_{\sigma;\delta}$, or a success, $(\mathtt{end})_{\sigma;\delta}$. In the first case, rule (ATRE), the transaction is aborted and started again from scratch. In the second case, we need to check if the log is consistent with the current evaluation state. A log is consistent if the read actions of δ can be performed on the current state. If the check fails, rule (ATFAIL), the transaction aborts. Otherwise, rule (ATOK), we commit the transaction: the names in $\textsc{rd}(\delta)$ are taken from the current state and a bunch of outputs on the names in $\textsc{wt}(\delta)$ are generated.

Reduction for Ongoing Expressions. Table 3 gives the semantics of ongoing atomic expressions. We recall that, in an expression $(\mathtt{rd}\,a.M)_{\sigma;\delta}$, the subscript σ is the *initial state*, that is a copy of the state at the time the block has been created and δ is the log of actions performed since the initiation of the transaction.

Rule (ARDOK) states that a read action $\mathtt{rd}\,a$ is recorded in the log δ if all the read actions in $\delta.\mathtt{rd}\,a$ can be performed in the initial state. If it is not the case, the ongoing expression fails, rule (ARDF). This test may be interpreted as a kind of optimization: if a transaction cannot commit in the initial state then, should it commit at the end of the atomic block, it would mean that the global state has been concurrently modified during the execution of the transaction. Note that we consider the initial state σ and not $\sigma \uplus \mathrm{WT}(\delta)$, which means that, in an atomic block, write actions are not directly visible (they cannot be consumed by a read action). This is coherent with the fact that outputs on $\mathrm{WT}(\delta)$ only take place after commit of the block. Rule (AWR) states that a write action always succeeds and is recorded in the current log.

The remaining rules govern the semantics of the `retry`, `end` and `orElse` constructs. These constructs are borrowed from the STM combinators used in the implementation of an STM system in Concurrent Haskell [17]. We define these operators with an equivalent semantics, with the difference that, in our case, a state is not a snapshot of the (shared) memory but a multiset of visible outputs. A composition M `orElse` N corresponds to the interleaving of the behaviors of M and N, which are independently evaluated with respect to the same evaluation state (but have distinct logs). The `orElse` operator is preemptive: the ongoing block M `orElse` N ends if and only M ends or M aborts and N ends.

3 Encoding Concurrency Primitives

Our first example is a simple solution to the celebrated *leader election* problem that does not yield to deadlock. Consider a system composed by n processes and a token, named t, that is modeled by an output $\overline{t}$. A process becomes a leader by getting (making an input on) t. As usual, all participants run the same process (except for the value of their identity). We suppose that there is only one copy of the token in the system and that leadership of process i is communicated to the other processes by outputting on a reserved name win_i. A participant that is not a leader outputs on $lose_i$. The protocol followed by the participants is defined by the following process:

$$L_i \triangleq \big(\mathtt{atom}\big(\mathtt{rd}\,t.\mathtt{wt}\,k.\mathtt{end}\;\mathtt{orElse}\;\mathtt{wt}\,k'.\mathtt{end}\big) \mid k.\overline{win_i} \mid k'.\overline{lose_i}\big)\backslash^0 k \backslash^0 k'$$

In this encoding, the atomic block is used to protect the concurrent accesses to t. If the process L_i commits its transaction and grabs the token, it immediately release an output on its private channel k. The transactions of the other participants may either fail or commit while releasing an output on their private channel k'. Then, the elected process L_i may proceed with a synchronization on k that triggers the output $\overline{win_i}$. The semantics of `atom()` ensures that only one transaction can acquire the lock and commit the atomic block, then no other process have acquired the token in the same round and we are guaranteed that there could be at most one leader.

This expressivity result is mixed blessing. Indeed, it means that any implementation of the atomic operator should be able to solve the leader election problem, which is known to be very expensive in the case of loosely-coupled systems or in presence of failures (see e.g. [21] for a discussion on the expressivity of process calculi and electoral systems). On the other hand, atomic transactions are optimistic and are compatible with the use of probabilistic approaches. Therefore it is still reasonable to expect a practical implementation of ATCCS.

In the following, we show how to encode two fundamental concurrency patterns, namely (preemptive versions of) the choice and join-pattern operators.

Guarded choice. We consider an operator for choice, $\mu_1.P_1 + \cdots + \mu_n.P_n$, such that every process is prefixed by an action μ_i that is either an output $\overline{a}_i$ or an input a_i. The semantics of choice is characterized by the following three reduction rules (we assume that Q is also a choice):

$$\text{(C-INP)}\ a.P+Q;\sigma\uplus\{a\} \rightarrow P;\sigma \qquad \text{(C-OUT)}\ \overline{a}.P+Q;\sigma \rightarrow P;\sigma\uplus\{a\}$$

$$\text{(C-PASS)}\quad \frac{a \notin \sigma \quad Q;\sigma \rightarrow Q';\sigma'}{a.P+Q;\sigma \rightarrow Q';\sigma'}$$

A minor difference with the behavior of the choice operator found in CCS is that our semantics gives precedence to the leftmost process (this is reminiscent of the preemptive behavior of `orElse`). Another characteristic is related to the asynchronous nature of the calculus, see rule (C-OUT): since an output action can always interact with the environment, a choice $\overline{a}.P+Q$ may react at once and release the process $\overline{a} \mid P$.

Like in the example of the leader election problem, we can encode a choice $\mu_1.P_1 + \cdots + \mu_n.P_n$ using an atomic block that will mediate the interaction with the actions $\mu_1,\ldots,\mu_n$. We start by defining a straightforward encoding of input/output actions into atomic actions: $[\![\overline{a}]\!] = \mathtt{wt}\,a$ and $[\![a]\!] = \mathtt{rd}\,a$. Then the encoding of choice is the process

$$[\![\mu_1.P_1+\cdots+\mu_n.P_n]\!] \triangleq \big(\mathtt{atom}\big([\![\mu_1]\!].[\![\overline{k}_1]\!].\mathtt{end}\ \mathtt{orElse}\ \cdots\ \mathtt{orElse}\ [\![\mu_n]\!].[\![\overline{k}_n]\!].\mathtt{end}\big)$$
$$\mid k_1.[\![P_1]\!] \mid \cdots \mid k_n.[\![P_n]\!]\big) \setminus^0 k_1 \ldots \setminus^0 k_n$$

The principle of the encoding is essentially the same that in our solution to the leader election problem. Actually, using the encoding for choice, we can rewrite our solution in the following form: $L_i \triangleq t.\overline{win_i} + \overline{lose_i}\,.\mathbf{0}$. Using the rules in Table 2, it is easy to see that our encoding of choice is compatible with rule (C-INP), meaning that:

$$\begin{aligned}
[\![a.P+Q]\!];\sigma\uplus\{a\} &\rightarrow^* \big(\{\!|(\mathtt{end})_{\sigma\uplus\{a\};\mathtt{rd}\,a.\mathtt{wt}\,k_1}\,|\!\}_M \mid k_1.[\![P]\!] \mid \ldots\big) \setminus^0 k_1 \setminus \ldots;\sigma\uplus\{a\}\\
&\rightarrow \big(\overline{k_1} \mid k_1.[\![P]\!] \mid \ldots\big) \setminus^0 k_1 \setminus \ldots;\sigma\\
&\rightarrow \big([\![P]\!] \mid \ldots\big) \setminus^0 k_1 \setminus \ldots;\sigma
\end{aligned}$$

where the processes in parallel with $[\![P]\!]$ are harmless. In the next section, we define a weak bisimulation equivalence $\approx_{\mathsf{a}}$ that can be used to garbage collect harmless processes in the sense that, e.g. $(P \mid k.Q) \setminus^0 k \approx_{\mathsf{a}} P$ if P has no occurrences of k. Hence, we could prove that $[\![a.P+Q]\!];\sigma\uplus\{a\} \rightarrow^* \approx_{\mathsf{a}} [\![P]\!];\sigma$, which is enough to show that our encoding is correct with respect to rule (C-INP). The same is true for rules (C-OUT) and (C-PASS).

Join Patterns. A multi-synchronization $(a_1 \times \cdots \times a_n).P$ may be viewed as an extension of input prefix in which communication requires a synchronization with the n outputs $\overline{a_1}, \ldots, \overline{a_n}$ at once. that is, we have the reduction:

$$\text{(J-INP)} \quad (a_1 \times \cdots \times a_n).P ; \sigma \uplus \{a_1, \ldots, a_n\} \;\rightarrow\; P ; \sigma$$

This synchronization primitive is fundamental to the definition of the Gamma calculus of Banâtre and Le Métayer and of the Join calculus of Fournet and Gonthier. It is easy to see that the encoding of a multi-synchronization (input) is a simple transaction:

$$[\![(a_1 \times \cdots \times a_n).P]\!] \triangleq \big(\mathtt{atom}([\![a_1]\!].\cdots.[\![a_n]\!].[\![\overline{k}]\!].\mathtt{end}) \mid k.[\![P]\!]\big) \setminus^0 k \quad \text{(where } k \text{ is fresh)}$$

and that we have $[\![(a_1 \times \cdots \times a_n).P]\!] ; \sigma \uplus \{a_1, \ldots, a_n\} \;\rightarrow^*\; \big(\mathbf{0} \mid [\![P]\!]\big) \setminus^0 k ; \sigma$, where the process $\big(\mathbf{0} \mid [\![P]\!]\big) \setminus^0 k$ is behaviorally equivalent to $[\![P]\!]$, that is:

$$[\![(a_1 \times \cdots \times a_n).P]\!] ; \sigma \uplus \{a_1, \ldots, a_n\} \;\rightarrow^* \approx_{\mathsf{a}}\; [\![P]\!] ; \sigma$$

Based on this encoding, we can define two interesting derived operators: a mixed version of multi-synchronization, $(\mu_1 \times \cdots \times \mu_n).P$, that mixes input and output actions; and a replicated version, that is analogous to replicated input.

$$[\![(\mu_1 \times \cdots \times \mu_n).P]\!] \triangleq \big(\mathtt{atom}([\![\mu_1]\!].\cdots.[\![\mu_n]\!].[\![\overline{k}]\!].\mathtt{end}) \mid k.[\![P]\!]\big) \setminus^0 k$$

$$[\![* (\mu_1 \times \cdots \times \mu_n).P]\!] \triangleq \big(\overline{r} \mid * r.\mathtt{atom}([\![\mu_1]\!].\cdots.[\![\mu_n]\!].[\![\overline{r}]\!].[\![\overline{k}]\!].\mathtt{end}) \mid * k.[\![P]\!]\big) \setminus^0 r \setminus^0 k$$

By looking at the possible reductions of these (derived) operators, we can define derived reduction rules. Assume δ is the log $[\![\mu_1]\!].\cdots.[\![\mu_n]\!]$, we have a simulation result comparable to the case for multi-synchronization, namely:

$$[\![(\mu_1 \times \cdots \times \mu_n).P]\!] ; \sigma \uplus \mathrm{RD}(\delta) \;\rightarrow^* \approx_{\mathsf{a}}\; [\![P]\!] ; \sigma \uplus \mathrm{WT}(\delta)$$

$$[\![* (\mu_1 \times \cdots \times \mu_n).P]\!] ; \sigma \uplus \mathrm{RD}(\delta) \;\rightarrow^* \approx_{\mathsf{a}}\; [\![* (\mu_1 \times \cdots \times \mu_n).P]\!] \mid [\![P]\!] ; \sigma \uplus \mathrm{WT}(\delta)$$

To obtain join-definitions, we only need to combine a sequence of replicated multi-synchronizations using the choice composition defined precedently. (We also need hiding to close the scope of the definition.) Actually, we can encode even more flexible constructs mixing choice and join-patterns. For the sake of simplicity, we only study examples of such operations. The first example is the (linear) join-pattern $(a \times b).P \wedge (a \times c).Q$, that may fire P if the outputs $\{a, b\}$ are in the global state σ and otherwise fire Q if $\{a, c\}$ is in σ (actually, real implementations of join-calculus have a preemptive semantics for pattern synchronization). The second example is the derived operator $(a \times b) + (b \times c \times \overline{a}).P$, such that P is fired if outputs on $\{a, b\}$ are available or if outputs on $\{b, c\}$ are available (in which case an output on a is also generated). These examples can be easily interpreted using atomic transactions:

$$\begin{aligned}
[\![(a \times b).P \wedge (a \times c).Q]\!] &\triangleq \big(\mathtt{atom}\big([\![a]\!].[\![b]\!].[\![\overline{k_1}]\!].\mathtt{end}\ \mathtt{orElse} \\
&\qquad\qquad [\![a]\!].[\![c]\!].[\![\overline{k_2}]\!].\mathtt{end}\big) \mid k_1.P \mid k_2.Q\big) \setminus^0 k_1 \setminus^0 k_2 \\
[\![(a \times b + b \times c \times \overline{a}).P]\!] &\triangleq \big(\mathtt{atom}\big([\![a]\!].[\![b]\!].[\![\overline{k}]\!].\mathtt{end}\ \mathtt{orElse} \\
&\qquad\qquad [\![b]\!].[\![c]\!].[\![\overline{a}]\!].[\![\overline{k}]\!].\mathtt{end}\big) \mid k.P\big) \setminus^0 k
\end{aligned}$$

In the next section we define the notion of bisimulation used for reasoning on the soundness of our encodings. We also define an equivalence relation for atomic expressions that is useful for reasoning on the behavior of atomic blocks.

4 Bisimulation Semantics

A first phase before obtaining a bisimulation equivalence is to define a Labeled Transition System (LTS) for ATCCS processes related to the reduction semantics.

Labeled Semantics of ATCCS. It is easy to derive labels from the reduction semantics given in Table 2. For instance, a reduction of the form $P;\sigma \rightarrow P';\sigma \uplus \{a\}$ is clearly an *output transition* and we could denote it using the transition $P \xrightarrow{\overline{a}} P'$, meaning that the effect of the transition is to add a message on a to the global state σ. We formalize the notion of label and transition. Besides output actions $\overline{a}$, which corresponds to an application of rule (OUT), we also need *block actions*, which are multisets of the form $\{a_1,\ldots,a_n\}$ corresponding to the commit of an atomic block, that is to the deletion of a bunch of names from the global state in rule (ATOK). Block actions include the usual labels found in LTS for CCS and are used for labeling input and communication transitions: an input action a, which intuitively corresponds to rules (IN) and (REP), is a shorthand for the (singleton) block action $\{a\}$; the silent action τ, which corresponds to rule (COM), is a shorthand for the empty block action $\emptyset$. In the following, we use the symbols $\theta,\gamma,\ldots$ to range over block actions and $\mu,\mu',\ldots$ to range over labels, $\mu ::= \overline{a} \mid \theta \mid \tau \mid a$.

The labeled semantics for ATCCS is the smallest relation $P \xrightarrow{\mu} P'$ satisfying the two following clauses:

1. we have $P \xrightarrow{\overline{a}} P'$ if there is a state σ such that $P;\sigma \rightarrow P';\sigma \uplus \{a\}$;
2. we have $P \xrightarrow{\theta} P'$ if there is a state σ such that $P;\sigma \uplus \theta \rightarrow P';\sigma$.

Note that, in the case of the (derived) action τ, we obtain from clause 2 that $P \xrightarrow{\tau} P'$ if there is a state σ such that $P;\sigma \rightarrow P';\sigma$. As usual, silent actions label transitions that do not modify the environment (in our case the global state) and so are invisible to an outside observer. Unlike CCS, the calculus has more examples of silent transition than mere internal synchronization, e.g. the initiation and evolution of an atomic block, see e.g. rules (ATST) and (ATPASS). Consequently, a suitable (weak) equivalence for ATCCS should not distinguish e.g. the processes atom(retry), atom(end), $(a.\overline{a})$ and $\mathbf{0}$. The same is true with input transitions. For instance, we expect to equate the processes $a.\mathbf{0}$ and atom(rd a .end).

Our labeled semantics for ATCCS is not based on a set of transition rules, as it is usually the case. Nonetheless, we can recover an axiomatic presentation of the semantics using the tight correspondence between labeled transitions and reductions characterized by Proposition 1.

Proposition 1. *Consider two processes P and Q. The following implications are true:*

(COM) *if* $P \xrightarrow{a} P'$ *and* $Q \xrightarrow{\overline{a}} Q'$ *then* $P \mid Q \xrightarrow{\tau} P' \mid Q'$;
(PAR) *if* $P \xrightarrow{\mu} P'$ *then* $P \mid Q \xrightarrow{\mu} P' \mid Q$ *and* $Q \mid P \xrightarrow{\mu} Q \mid P'$;

(HID) *if $P \xrightarrow{\mu} P'$ and the name a does not appear in μ then $P \setminus^n a \xrightarrow{\mu} P' \setminus^n a$;*
(HIDOUT) *if $P \xrightarrow{\overline{a}} P'$ then $P \setminus^n a \xrightarrow{\tau} P' \setminus^{n+1} a$;*
(HIDAT) *if $P \xrightarrow{\mu} P'$ and $\mu = \theta \uplus \{a^m\}$, where a is a name that does not appear in the label θ, then $P \setminus^{n+m} a \xrightarrow{\theta} P' \setminus^n a$.*

Proof. In each case, we have a transition of the form $P \xrightarrow{\mu} P'$. By definition, there are states σ and σ' such that $P;\sigma \to P';\sigma'$. The property is obtained by a simple induction on this reduction (a case analysis on the last reduction rule is enough). □

We define additional transition relations used in the remainder of the paper. As usual, we denote by $\Rightarrow$ the *weak transition relation*, that is the reflexive and transitive closure of $\xrightarrow{\tau}$. We denote by $\overset{\mu}{\Rightarrow}$ the relation $\Rightarrow$ if $\mu = \tau$ and $\Rightarrow \xrightarrow{\mu} \Rightarrow$ otherwise. If s is a sequence of labels $\mu_0 \ldots \mu_n$, we denote $\xrightarrow{s}$ the relation such that $P \xrightarrow{s} P'$ if and only if there is a process Q such that $P \xrightarrow{\mu_0} Q$ and $Q \xrightarrow{\mu_1 \cdots \mu_n} P'$ (and $\xrightarrow{s}$ is the identity relation when s is the empty sequence ε). We also define a weak version $\overset{s}{\Rightarrow}$ of this relation in the same way. Lastly, we denote $\xrightarrow{a^n}$ the relation $\xrightarrow{a} \ldots \xrightarrow{a}$, the composition of n copies of $\xrightarrow{a}$.

Asynchronous Bisimulation for Processes and Expressions. Equipped with a labeled transition system, we can define a weak *asynchronous bisimulation* relation, denoted $\approx_{\mathsf{a}}$, in the style of [2].

Definition 1 (weak asynchronous bisimulation). *A symmetric relation $\mathcal{R}$ is a weak asynchronous bisimulation if whenever $P \mathcal{R} Q$ then the following holds:*

1. *if $P \xrightarrow{\overline{a}} P'$ then there is Q' such that $Q \overset{\overline{a}}{\Rightarrow} Q'$ and $P' \mathcal{R} Q'$;*
2. *if $P \xrightarrow{\theta} P'$ then there is a process Q' and a block action γ such that $Q \overset{\gamma}{\Rightarrow} Q'$ and $\left(P' \mid \prod_{a \in (\gamma \setminus \theta)} \overline{a}\right) \mathcal{R} \left(Q' \mid \prod_{a \in (\theta \setminus \gamma)} \overline{a}\right)$.*

We denote with $\approx_{\mathsf{a}}$ the largest weak asynchronous bisimulation.

Assume $P \approx_{\mathsf{a}} Q$ and $P \xrightarrow{\tau} P'$, the (derived) case for silent action entails that there is Q' and θ such that $Q \overset{\theta}{\Rightarrow} Q'$ and $P' \mid \prod_{a \in \theta} \overline{a} \approx_{\mathsf{a}} Q'$. If θ is the silent action, $\theta = \{\,\}$, we recover the usual condition for bisimulation, that is $Q \Rightarrow Q'$ and $P' \approx_{\mathsf{a}} Q'$. If θ is an input action, $\theta = \{a\}$, we recover the definition of asynchronous bisimulation of [2]. Due to the presence of block actions γ, the definition of $\approx_{\mathsf{a}}$ is slightly more complicated than in [2], but it is also more compact (we only have two cases) and more symmetric. Hence, we expect to be able to reuse known methods and tools for proving the equivalence of ATCCS processes. Another indication that $\approx_{\mathsf{a}}$ is a good choice for reasoning about processes is that it is a congruence.

Theorem 1. *Weak asynchronous bisimulation $\approx_{\mathsf{a}}$ is a congruence.*

Proof. It suffices to prove that $\approx_{\mathsf{a}}$ is preserved by every operator of the calculus [1]. □

We need to define a specific equivalence relation to reason on transactions. Indeed, the obvious choice that equates two expressions M and N if $\mathtt{atom}(M) \approx_{\mathsf{a}} \mathtt{atom}(N)$ does not lead to a congruence. For instance, we have $(\mathtt{rd}\,a.\mathtt{wt}\,a.\mathtt{end})$ equivalent to $\mathtt{end}$ while $\mathtt{atom}(\mathtt{rd}\,a.\mathtt{wt}\,a.\mathtt{end}\ \mathtt{orElse}\ \mathtt{wt}\,b.\mathtt{end}) \not\approx_{\mathsf{a}} \mathtt{atom}(\mathtt{end}\ \mathtt{orElse}\ \mathtt{wt}\,b.\mathtt{end})$. The first transaction may output a message on b while the second always end silently.

Table 4. Algebraic Laws of Transactions

Laws for atomic expressions:			
(COMM)	$\alpha.\beta.M$	$\simeq$	$\beta.\alpha.M$
(DIST)	$\alpha.(M\ \texttt{orElse}\ N)$	$\simeq$	$(\alpha.M)\ \texttt{orElse}\ (\alpha.N)$
(ASS)	$M_1\ \texttt{orElse}\ (M_2\ \texttt{orElse}\ M_3)$	$\simeq$	$(M_1\ \texttt{orElse}\ M_2)\ \texttt{orElse}\ M_3$
(IDEM)	$M\ \texttt{orElse}\ M$	$\simeq$	M
(ABSRT1)	$\alpha.\texttt{retry}$	$\simeq$	$\texttt{retry}$
(ABSRT2)	$\texttt{retry}\ \texttt{orElse}\ M$	$\simeq$	$M \simeq M\ \texttt{orElse}\ \texttt{retry}$
(ABSEND)	$\texttt{end}\ \texttt{orElse}\ M$	$\simeq$	$\texttt{end}$
Laws for processes:			
(ASY)	$a.\overline{a}$	$\approx_{\mathsf{a}}$	$\mathbf{0}$
(A-ASY)	$\texttt{atom}(\texttt{rd}\,a.\texttt{wt}\,a.\texttt{end})$	$\approx_{\mathsf{a}}$	$\mathbf{0}$
(A-1)	$\texttt{atom}(\texttt{rd}\,a.\texttt{end})$	$\approx_{\mathsf{a}}$	$a.\mathbf{0}$

We define an equivalence relation between atomic expressions $\simeq$, and a *weak atomic preorder* $\sqsupseteq$, that relates two expressions if they end (or abort) for the same states. We also ask that equivalent expressions should perform the same changes on the global state when they end. We say that two logs δ, δ' have same effects, denoted $\delta =_\sigma \delta'$ if $\sigma \setminus \mathrm{RD}(\delta) \uplus \mathrm{WT}(\delta) = \sigma \setminus \mathrm{RD}(\delta') \uplus \mathrm{WT}(\delta')$. We say that $M \sqsupseteq_\sigma N$ if and only if either (1) $(N)_{\sigma;\varepsilon} \Rightarrow (\texttt{retry})_{\sigma,\delta}$; or (2) $(N)_{\sigma;\varepsilon} \Rightarrow (\texttt{end})_{\sigma,\delta}$ and $(M)_{\sigma;\varepsilon} \Rightarrow (\texttt{end})_{\sigma;\delta'}$. Similarly, we have $M \simeq_\sigma N$ if and only if either (1) $(M)_{\sigma;\varepsilon} \Rightarrow (\texttt{retry})_{\sigma,\delta}$ and $(N)_{\sigma;\varepsilon} \Rightarrow (\texttt{retry})_{\sigma,\delta'}$; or (2) $(M)_{\sigma;\varepsilon} \Rightarrow (\texttt{end})_{\sigma;\delta}$ and $(N)_{\sigma;\varepsilon} \Rightarrow (\texttt{end})_{\sigma,\delta'}$ with $\delta =_\sigma \delta'$.

Definition 2 (weak atomic equivalence). *Two atomic expressions M, N are equivalent, denoted $M \simeq N$, if and only if $M \simeq_\sigma N$ for every state σ. Similarly, we have $M \sqsupseteq N$ if and only if $M \sqsupseteq_\sigma N$ for every state σ.*

While the definition of $\sqsupseteq$ and $\simeq$ depend on a universal quantification over states, testing the equivalence of two expressions is not expensive. First, we can rely on a monotonicity property of reduction: if $\sigma \subseteq \sigma'$ then for all M the effect of $(M)_{\sigma,\delta}$ is included in those of $(M)_{\sigma',\delta}$. Moreover, we define a normal form for expressions later in this section (see Proposition 2) that greatly simplifies the comparison of expressions. Another indication that $\simeq$ is a good choice of equivalence for atomic expressions is that it is a congruence.

Theorem 2. *Weak atomic equivalence $\simeq$ is a congruence.*

On the Algebraic Structure of Transactions. The equivalence relations $\simeq$ and $\approx_{\mathsf{a}}$ can be used to prove interesting laws of atomic expressions and processes. We list some of these laws in Table 4. Let $\mathcal{M}$ denotes the set of all atomic expressions. The behavioral rules for atomic expressions are particularly interesting since they exhibit a rich algebraic structure for $\mathcal{M}$. For instance, rules (COMM) and (DIST) state that action prefix $\alpha.M$ is a commutative operation that distribute over `orElse`. We also have that $(\mathcal{M}, \texttt{orElse}, \texttt{retry})$ is an idempotent semigroup with left identity `retry`, rules (ASS), (ABSRT2) and (IDEM), and that `end` annihilates $\mathcal{M}$, rule (ABSEND). Most of these laws appear in [17] but are not formally proved.

Actually, we can show that the structure of $\mathcal{M}$ is close to that of a bound join-semilattice. We assume unary function symbols $a()$ and $\overline{a}()$ for every name a (a term $\overline{a}(M)$ is intended to represent a prefix $\texttt{wt}\,a.M$) and use the symbols $\sqcup, \texttt{1}, \texttt{0}$ instead of `orElse`, `end`, `retry`. With this presentation, the behavioral laws for atomic expression are almost those of a semilattice. By definition of $\sqsupseteq$, we have that $M \sqcup M' \backsimeq M$ if and only if $M \sqsupseteq M'$ and for all M,N we have $\texttt{1} \sqsupseteq M \sqcup N \sqsupseteq M \sqsupseteq \texttt{0}$.

$$\mu(\mu'(M)) \backsimeq \mu'(\mu(M)) \qquad \mu(M \sqcup N) \backsimeq \mu(M) \sqcup \mu(N) \qquad \mu(\texttt{0}) \backsimeq \texttt{0}$$

$$\texttt{0} \sqcup M \backsimeq M \backsimeq M \sqcup \texttt{0} \qquad \texttt{1} \sqcup M \backsimeq \texttt{1}$$

It is possible to prove other behavioral laws to support our interpretation of `orElse` as a join. However some important properties are missing, most notably, while $\sqcup$ is associative, it is not commutative. For instance, $a(\overline{b}(\texttt{1})) \sqcup \texttt{1} \not\backsimeq \texttt{1}$ while $\texttt{1} \backsimeq \texttt{1} \sqcup a(\overline{b}(\texttt{1}))$, rule (ABSEND). This observation could help improve the design of the transaction language: it will be interesting to enrich the language so that we obtain a real lattice.

Normal Form for Transactions. Next, we show that it is possible to rearrange an atomic expression (using behavioral laws) to put it into a simple *normal form*. This procedure can be understood as a kind of compilation that transform an expression M into a simpler form.

Informally, an atomic expression M is said to be in *normal form* if it does not contain nested `orElse` (all occurrences are at top level) and if there are no redundant branches. A redundant branch is a sequence of actions that will never be executed. For instance, the read actions in $\texttt{rd}\,a.\texttt{end}$ are included in $\texttt{rd}\,a.\texttt{rd}\,b.\texttt{end}$, then the second branch in the composition $(\texttt{rd}\,a.\texttt{end})$ `orElse` $(\texttt{rd}\,a.\texttt{rd}\,b.\texttt{end})$ is redundant: obviously, if $\texttt{rd}\,a.\texttt{end}$ fails then $\texttt{rd}\,a.\texttt{rd}\,b.\texttt{end}$ cannot succeed. We overload the functions defined on logs and write $\mathrm{RD}(M)$ for the (multiset of) names occurring in read actions in M. We define $\mathrm{WT}(M)$ similarly. In what follows, we abbreviate $(M_1\ \texttt{orElse}\ \ldots\ \texttt{orElse}\ M_n)$ with the expression $\bigsqcup_{i\in 1..n} M_i$. We say that an expression M is in *normal form* if it is of the form $\bigsqcup_{i\in 1..n} K_i$ where for all indexes $i,j \in 1..n$ we have: (1) K_i is a sequence of action prefixes $\alpha_{j_1}.\ldots.\alpha_{j_{n_i}}.\texttt{end}$; and (2) $\mathrm{RD}(K_i) \not\subseteq \mathrm{RD}(K_j)$ for all $i < j$. Condition (1) requires the absence of nested `orElse` and condition (2) prohibits redundant branches (it also means that all branches, but the last one, has a read action).

Proposition 2. *For every expression M there is a normal form M' such that $M \backsimeq M'$.*

Proof. Laws (COMM), (DIST) and (ASS) in Table 4 can be applied for eliminating nested `orElse`. Next, we use the fact that if K is a redundant branch of M then $M \sqsupseteq K$. □

Our choice of using bisimulation for reasoning about atomic transactions may appear arbitrary. In the long version [1], we study a testing equivalence for ATCCS, more particularly an asynchronous may testing semantics [15].

5 Future and Related Works

There is a long history of works that try to formalize the notions of transactions and atomicity, and a variety of approaches to tackle this problem. We review some of these works that are the most related to ours.

We can list several works that combine ACID transactions with process calculi. Gorrieri et al [16] have modeled concurrent systems with atomic behaviors using an extension of CCS. They use a two-level transition systems (a high and a low level) where high actions are decomposed into atomic sequences of low actions. To enforce isolation, atomic sequences must go into a special invisible state during all their execution. Contrary to our model, this work does not follow an optimistic approach: sequences are executed sequentially, without interleaving with other actions, as though in a critical section. Another related calculus is RCCS, a reversible version of CCS [13,14] based on an earlier notion of process calculus with backtracking [4]. In RCCS, each process has access to a log of its synchronization's history and may always wind back to a previous state. This calculus guarantees the ACD properties of transactions (isolation is meaningless since RCCS do not use a shared memory model). Finally, a framework for specifying the semantics of transactions in an object calculus is given in [23]. The framework is parametrized by the definition of a transactional mechanism and allows the study of multiple models, such as the usual lock-based approach. In this work, STM is close to a model called *versioning semantics*. Like in our approach, this model is based on the use of logs and is characterized by an optimistic approach where log consistency is checked at commit time. Fewer works consider behavioral equivalences for transactions. A foundational work is [6], that gives a theory of transactions specifying atomicity, isolation and durability in the form of an equivalence relation on processes, but it provides no formal proof system.

Linked to the upsurge of works on Web Services (and on long running Web transactions), a larger body of works is concerned with formalizing *compensating transactions*. In this context, each transactive block of actions is associated with a compensation (code) that has to be run if a failure is detected. The purpose of compensation is to undo most of the visible actions that have been performed and, in this case, atomicity, isolation and durability are obviously violated. We give a brief survey of works that formalize compensable processes using process calculi. These works are of two types: (1) *interaction based compensation* [7,8,19], which are extensions of process calculi (like π or join-calculus) for describing transactional choreographies where composition take place dynamically and where each service describes its possible interactions and compensations; (2) *compensable flow composition* [9,11], where ad hoc process algebras are designed from scratch to describe the possible flow of control among services. These calculi are oriented towards the orchestration of services and service failures. This second approach is also followed in [3,5] where two frameworks for composing transactional services are presented.

The study of ATCCS is motivated by our objective to better understand the semantics of the STM model. Obtaining a suitable behavioral equivalence for atomic expression is a progress for the verification of concurrent applications that use STM. However, we can imagine using our calculus for other purposes. An interesting problem is to develop an approach merging atomic and compensating transactions. A first step in this direction is to enrich our language and allow the parallel composition of atomic expressions and the nesting of transactions. We are currently working on this problem. Another area for research stems from our observation (see Section 4) that the algebraic structure of atomic expressions is lacking interesting property. Indeed, it will be interesting to enrich the language of expressions in order to obtain a real lattice. The addition of a symmetric choice operator for atomic expressions may be a solution, but it could introduce unwanted nondeterminism in the evaluation of transactions.

References

1. L. Acciai, M. Boreale and S. Dal Zilio. A Concurrent Calculus with Atomic Transactions (long version). `http://arxiv.org/abs/cs.LO/0610137`.
2. R. Amadio, I. Castellani and D. Sangiorgi. On Bisimulations for the Asynchronous π-Calculus. *Th. Comp. Sci.*, 195(2):291–324, 1998.
3. D. Berardi, D. Calvanese, G. De Giacomo, R. Hull and M. Mecella. Automatic Composition of Transition-Based Web Services with Messaging. In *Proc. of VLDB*, 2005.
4. J.A. Bergstra, A. Ponse and J.J. van Wamel. Process Algebra with Backtracking. In *Proc. of REX Workshop*, LNCS 803, 1994.
5. S. Bhiri, O. Perrin and C. Godart. Ensuring Required Failure Atomicity of Composite Web Services. In *Proc. of WWW*, ACM Press, 2005.
6. A.P. Black, V. Cremet, R. Guerraoui and M. Odersky. An Equational Theory for Transactions. In *Proc. of FSTTCS*, LNCS 2914, 2003.
7. L. Bocchi, C. Laneve and G. Zavattaro. A Calculus for Long Running Transactions. In *Proc. of FMOODS*, LNCS 2884, 2003.
8. R. Bruni, H.C. Melgratti and U. Montanari. Nested Commits for Mobile Calculi: extending Join. In *Proc. of IFIP TCS*, 563–576, 2004.
9. R. Bruni, H.C. Melgratti and U. Montanari. Theoretical Foundations for Compensations in Flow Composition Languages. In *Proc. of POPL*, ACM Press, 209–220, 2005.
10. N. Busi, R. Gorrieri, G. Zavattaro. A Process Algebraic View of Linda Coordination Primitives. *Th. Comp. Sci.*, 192(2):167–199, 1998.
11. M.J. Butler, C. Ferreira and M.Y. Ng. Precise Modeling of Compensating Business Transactions and its Application to BPEL. In *J. UCS*, 11:712–743, 2005.
12. T. Chothia and D. Duggan. Abstractions for Fault-Tolerant Global Computing. *Th. Comp. Sci.*, 322(3):567–613, 2004.
13. V. Danos and J. Krivine. Reversible Communicating System. In *Proc. of CONCUR*, LNCS 3170, 2004.
14. V. Danos and J. Krivine. Transactions in RCCS. In *Proc. of CONCUR*, LNCS 3653, 2005.
15. R. De Nicola and M.C.B. Hennessy. Testing Equivalence for Processes. *Th. Comp. Sci.*, 34:83–133, 1984.
16. R. Gorrieri, S. Marchetti and U. Montanari. A^2CCS: Atomic Actions for CCS. *Th. Comp. Sci.*, 72(2-3):203–223, 1990.
17. T. Harris, S. Marlow, S.P. Jones and M. Herlihy. Composable Memory Transactions. In *Proc. of PPOPP*, ACM Press, 48–60, 2005.
18. M. Herlihy, J.E. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures In *Proc. of International Symposium on Computer Architecture*, 1993.

19. C. Laneve and G. Zavattaro. Foundations of Web Transactions. In *Proc. of FoSSaCS*, LNCS 3441, 2005.
20. R. Milner. Calculi for Synchrony and Asynchrony. *Th. Comp. Sci.*, 25:267–310, 1983.
21. C. Palamidessi. Comparing the Expressive Power of the Synchronous and the Asynchronous pi-calculus. *Math. Struct. in Comp. Sci.*, 13(5), 2003.
22. N. Shavit and D. Touitou. Software Transactional Memory. In *Proc. of Principles of Distributed Computing*, ACM Press, 1995.
23. J. Vitek, S. Jagannathan, A. Welc and A.L. Hosking. A semantic Framework for Designer Transactions. In *Proc. of ESOP*, LNCS 2986, 2004.