An Equational Theory for Transactions ⋆

Andrew P. Black[1], Vincent Cremet[2], Rachid Guerraoui[2], and Martin Odersky[2]

[1] OGI School of Science & Engineering, Oregon Health and Science University
andrew.black@ogi.edu
[2] Ecole polytechnique Federale de Lausanne (EPFL)
{vincent.cremet, rachid.guerraoui, martin.odersky}@epfl.ch

Abstract. Transactions are commonly described as being ACID: All-or-nothing, Consistent, Isolated and Durable. However, although these words convey a powerful intuition, the ACID properties have never been given a precise semantics in a way that disentangles each property from the others. Among the benefits of such a semantics would be the ability to trade-off the value of a property against the cost of its implementation. This paper gives a sound equational semantics for the transaction properties. We define three categories of actions, A-actions, I-actions and D-actions, while we view Consistency as an induction rule that enables us to derive system-wide consistency from local consistency. The three kinds of action can be nested, leading to different forms of transactions, each with a well-defined semantics. Conventional transactions are then simply obtained as ADI-actions. From the equational semantics we develop a formal proof principle for transactional programs, from which we derive the induction rule for Consistency.

1 Introduction

Failure, or rather partial failure, is one of the most complex issues in computing. By definition, a failure occurs when some component violates its specification: it has "gone wrong" in some serious but unspecified manner, and therefore reasoning about it means reasoning about an unknown state. To cope with such a situation we use abstractions that provide various kinds of "failure-tight compartment": like water-tight doors on a ship, they keep the computation afloat and give us a dry place to stand while we try to understand what has gone wrong and what can be done about it. The familiar notion of address space in an operating system is one such: the address space boundary limits the damage that can be done by a misbehaving program, and gives us some confidence that the damage has not spread to programs in other address spaces.

Transactions. The most successful abstraction for coping with failure is the transaction, which has emerged from earlier notions of atomic action [1]. The

⋆ This research was partially supported by the National Science Foundation of the USA under awards CCR-0098323 and CDA-9703218, by DARPA under the PCES program, and by the European Union, Project PEPITO, IST-2001-33234.

most popular characterization of transactions is due to Haerder and Reuter [2], who coined the term ACID to describe their four essential properties.

The "A" in ACID stands for all-or-nothing; it means that a transaction either completes or has no effect. In other words, despite failures, the transaction never produces partial effects. "I" stands for isolation; it means that the intermediate states of data manipulated by a transaction are not visible to any other committed transaction, *i.e.*, to any transaction that completes. "D" stands for durability; it means that the effects of a committed transaction are not undone by a failure. "C" stands for consistency; the C property has a different flavour from the other properties because part of the responsibility for maintaining consistency remains with the programmer of the transaction. In contrast, all-or-nothing, isolation and durability are the system's responsibility. Let us briefly explore this distinction.

Consistency is best understood as a contract between the programmer writing individual transactions and the system that implements them. Roughly speaking, the contract is the following: if the programmer ensures the consistency of every individual transaction, and also ensures that the initial state is consistent, then the system will ensure that consistency applies *globally* and *forever*, despite concurrency and failure. Consistency is thus like an induction axiom: it reduces the problem of maintaining the consistency of the whole of a concurrent system subject to failure to the much simpler problem of maintaining the consistency of a series of failure-free sequential actions. One of the main results of this paper is to state and prove this folklore idea for the first time in a formal way (see Theorem 2 in section 4).

The "ACID" formulation of transactions has been current for twenty years. During that time transactions have evolved, and have spawned variants such as nested transactions [3] and distributed transactions [4]. Yet, as far as we are aware, there has been no successful formalization of exactly what the A, D and I properties mean *individually*. In this paper we present such a formalization.

We believe that this work has value beyond mere intellectual interest. First, various kinds of atomic, durable and isolated actions are routinely used by the systems community; this work illuminates the relationship of these abstractions to the conventional transactions (T-actions) of the database world. Second, we have hopes that by separating out the *semantics* of A-, I- and D-actions, and showing how they can be composed to create T-actions, we will pave the way for the creation of separate *implementations* of A-, I- and D-actions, and their composition to create *implementations* of T-actions. To date, the implementations of the three properties have been interdependent. For example, implementing isolation using timestamp concurrency control techniques rather than locking changes the way that durability must be implemented.

Background. Transactions promise the programmer that he will never see a failure. In a 1991 paper [5], Black attempted to capture this promise using equivalence rules. Translated into a notation consistent with that used in the remainder

of this paper, the rules for all-or-nothing (1) and durability (2) were

$$\langle a \rangle \;||\; \downarrow\uparrow \;\equiv\; \langle a \rangle + \mathsf{skip} \tag{1}$$

$$\langle a \rangle \;;\; \downarrow\uparrow \;\equiv\; \langle a \rangle \tag{2}$$

where $\langle a \rangle$ represents a transaction that executes the command a, $||$ represents parallel composition, $\downarrow\uparrow$ represents a failure, and skip is the null statement. The alternation operator $+$ represents non-deterministic choice, so the right-hand side of (1) is a program that either makes the same state transformation as $\langle a \rangle$, or as skip — but we cannot *a priori* know which. The intended meaning of (1) was that despite failures, the effect of the transaction $\langle a \rangle$ would either be *all* of $\langle a \rangle$ or *nothing* (skip). Similarly, equation (2) says that a failure occurring after a transaction has committed will have no effect on that transaction.

These rules are not independent, however. For example, relaxing the all-or-nothing rule changes equation (2) as well as equation (1). It also turns out that $\downarrow\uparrow$ cannot be treated as a process (see section 2). Thus, the main contribution of the 1991 paper was the identification of an interesting problem, not its solution. The solution was not trivial: it has taken another twelve years!

The reader might ask *why* skip is one of the possible meanings of $\langle a \rangle \;||\; \downarrow\uparrow$. The answer is that in practice this is the price that we must pay for atomicity: the only way that an implementation can guarantee all-or-nothing, in a situation where an unexpected failure makes it impossible to give us all, is to give us nothing.

Implementations have taken advantage of this equivalence to abort a few transactions even when no failure actually occurs: even though it might have been possible to commit the transaction, the implementation might decide that it is inconvenient or too costly. For example, in systems using optimistic concurrency control, a few logically unnecessary aborts are considered to be an acceptable price to pay for increased throughput.

In some centralized systems failure is rare enough that it may be acceptable to abort all active transactions when a failure does occur. However, in a distributed system, failure is commonplace: it is not acceptable to abort every computation in a system (for example, in the Internet) because one data object has become unavailable. Thus, we need to be able to reason about *partial* failures.

Related work. Transactions originated in the database arena [6] and were eventually ported to distributed operating-systems like Tabs [7] and Camelot [8], as well as to distributed programming languages like Argus [9], Arjuna [10], Avalon [11], KAROS [12] or Venari [13]. During this phase, transaction mechanisms began to be "deconstructed" into simpler parts. The motivation was to give the programmer the ability to select — and pay for — a subset of the transaction properties. However, to our knowledge there was no attempt to define precisely what each property guaranteed, or how the properties could be combined.

The "Recoverable Virtual Memory" abstraction of the Camelot system is an example of a less-than-ACID transaction. Recoverable Memory Memory supported two kinds of transactions: all-or-nothing transactions (called "no-flush"

transactions), and all-or-nothing transactions that are also durable. Concurrency-control was factored out into a separate mechanism that the programmer could use to ensure isolation. This gave the programmer a very flexible way of trading-off transaction properties for efficient implementations, but the meaning of the various properties was not rigorously defined and it is not clear what guarantees their combination would enjoy. Similarly, in Venari, the programmer can easily define durable transactions, atomic transactions and isolated transactions, but the meaning of the combinations of these properties was not defined formally.

Our equational semantics provide a way to reason about individual properties of less-than-ACID transactions and about the meaning of their composition.

The ACTA formalism [14] was introduced to capture the functionalities of various transactional models. In particular, the aim was to allow the specification of significant events beyond commit and abort (useful for long-lived transactions) and to allow the specification of arbitrary transaction structures in terms of dependencies between transactions (read-from relations). The notation enabled one to informally describe various transaction models, such as open and nested transactions, but did not attempt to capture the precise meaning of the individual transaction properties, nor was it used to study their composition.

Interestingly, all modern transactional platforms we know of, including ArjunaTS [15], BEA Weblogics [16], IBM Webspheres [17], Microsoft MTS [18], and Sun EJB [19] provide the programmer with the ability to select the best variant of transactional semantics for a given application. Our equational semantics might provide a sound theoretical framework to help make the appropriate choice.

In [20] the authors also formalize the concepts of crash and recovery by extending the π-calculus. They are able to prove correctness of the two-phase commit protocol, which can be used to implement transactions, but they do not model full transactions nor a fortiori give a consistency preservation theorem as we do. Contrary to our work which views serializability as the only meaning of the isolation property, [21] defines the notion of *semantic correctness*: a schedule is considered correct if its *effect* is the same as some serializable schedule, and not only if it is serializable (the concept of effect is modeled using Hoare logic).

Overview Our technical treatment is organized as follows. We first define a syntax for *pre-processes*, in which actions are combined using sequential, parallel and non-deterministic composition. With a simple kind system, we select from these pre-processes a set of *well-formed processes*. We then present a set of axioms that define an equivalence relation on well-formed processes and discuss the rationale underlying these axioms.

By turning the axioms into a rewriting system modulo some structural equalities, we prove that every process has a unique *canonical form* (Theorem 1 ; sec. 4). Canonical forms contain neither embedded failures nor parallel composition, and thus they allow us to capture the semantics of an arbitrary process in a simple way.

If we restrict further the shape of a process so that it is built exclusively from *locally consistent sequences*, then we can prove that its canonical form is

also built from consistent sequences (Theorem 2 ; sec. 4). Hence, we can show that in our model of transactions, local consistency implies global consistency.

For space reasons this document does not contain the proofs of theorems. They can be found in a compagnon document [22] together with more detailed explanations about this work.

2 Processes

We start with the definition of several interesting classes of processes.

As is usually the case for programming languages, we introduce the set of processes in two stages: we first define a set of *syntactically* correct objects that we call *pre-processes* and we then consider a *semantically* correct subset whose elements are called *well-formed* processes or, more briefly, processes.

The syntax of pre-processes is as follows.

$P, Q ::= a, b, c, \ldots$		primitive action
	$\mid \langle P \rangle_A$	all-or-nothing action
	$\mid \langle P \rangle_D$	durable action
	$\mid \langle P \rangle_I$	isolated action
	$\mid P \,;\, Q$	sequential composition
	$\mid P \mid\mid Q$	parallel composition
	$\mid P + Q$	non-deterministic choice
	$\mid \mathsf{skip}$	null action
	$\mid \mathit{crash}(P)$	one or more crashes and recoveries during P

The operators have the following precedence: ; binds more than || which binds more tightly than +.

Primitive actions. A primitive action represents an access to a shared resource. A typical primitive action might be the invocation of a method on a global object. We use a different symbol $a, b, \ldots$ for each primitive action.

Decomposed transactions. Three kinds of brackets are used to group actions:

$\langle P \rangle_A$ processes are either executed completely or not at all.
$\langle P \rangle_I$ processes are *isolated*; a parallel execution of such processes has always same effect as some sequential execution of the same processes.
$\langle P \rangle_D$ processes are *durable*; once completed, their effect cannot be undone by subsequent failures.

There is no fourth kind of bracket for consistent actions; as discussed in Section 1, consistency is a meta-property that needs to be established by the programmer for individual sequences of actions. Also missing is a kind of bracket corresponding to classical, full-featured transactions. There is no need for it, since such transactions can be expressed by a nesting of all-or-nothing, durable, and isolated actions. That is, we will show that the process $\langle\langle\langle P \rangle_A\rangle_D\rangle_I$ represents P executed as a classical transaction.

Formal reasoning about failures requires that we delimit their scope. In our calculus we use the action brackets for this purpose also. Thus, a crash/recovery event inside an action should be interpreted as a crash/recovery event that is local to the memory objects accessible from that action. For instance, in the action $\langle\langle P\rangle_I \;||\; \langle Q \;;\; \downarrow\uparrow\rangle_I\rangle_A$, the crash/recovery will affect only the nested action containing Q, and not the action containing P nor the top-level action. In contrast, a crash/recovery event occurring in some action P will in general affect actions nested inside P.

Failures. The term $crash(P)$ represents the occurrence of one or more crash/-recovery events during the execution of process P. Each crash/recovery event can be thought of as an erasure of the volatile local memory of the process P (the crash) followed by the reinitialization of that memory from the durable backup (the recovery).

We represent failures that occur independently of any other action as if they occurred during a null action. We use the following shorthand notation to represent such a single failure event:

$$\downarrow\uparrow \equiv crash(\mathsf{skip})$$

One might wonder why we did not instead start by taking the symbol $\downarrow\uparrow$ as a primitive and letting $crash(P)$ be an abbreviation for $\downarrow\uparrow \;||\; P$. This was in fact our initial approach, but we encountered problems that led us to the interesting observation that crash/recovery is not an isolated action and that it cannot therefore be composed in parallel with other processes. This is explained in more detail in Section 2.

Note also that we consider a crash/recovery to be atomic. This means that we do not permit the crash phase to be dissociated from the recovery phase. We exclude, for instance, the possibility that another action can occur between a crash and its associated recovery.

Well-formed processes . We have a very simple notion of well-formedness in our calculus. The only restriction that we impose is that a process which is to be executed in parallel with others must be *interleavable*. Roughly speaking, an interleavable process is a term of the calculus that is built at the outermost level from actions enclosed in isolation brackets. These brackets define the grain of the interleaving.

We define the set of (well-formed) processes and the set of interleavable processes by mutual induction. The grammar of well-formed processes is almost the same as the grammar for pre-processes except that now parallel composition is allowed only for interleavable processes.

An important property of well-formed processes is that $crash(P)$ is not interleavable. So, for example,

$$\langle P\rangle_I \;||\; \langle Q\rangle_I \;||\; \downarrow\uparrow$$

is not a well-formed process. We exclude this process for the following reason: seen by itself, $\langle P \rangle_I \parallel \langle Q \rangle_I$ is equivalent to some serialization of P and Q — either $\langle P \rangle_I \,;\, \langle Q \rangle_I$ or $\langle Q \rangle_I \,;\, \langle P \rangle_I$. Applying a similar serialization law to the $\downarrow\uparrow$ component, one could be tempted to conclude that the crash will happen possibly during P or during Q, but certainly not during both P and Q. However, such a conclusion is clearly too restrictive, since it excludes every scheme for implementing transactions in parallel, and admits as the only possible implementations those which execute all isolated actions in strict sequence.

Canonical processes. Informally, a process in canonical form consists of a nondeterministic choice of one or more alternatives. Each alternative might start with an optional crash/recovery and is then followed by a sequence of primitive actions, possibly nested inside atomic, isolated or durable brackets. Note that a crash recovery at the very beginning of a process has no observable effect, as there are no actions that can be affected by it. The existence of canonical forms gives us an important *proof principle* for transactions. To prove a property $\mathcal{P}(P)$ of some process P, we transform P into an equivalent process C in canonical form, and prove instead $\mathcal{P}(C)$. The latter is usually much easier than the former, since processes in canonical form contain neither embedded failures nor parallel compositions.

Locally consistent processes. We now define a class of well-formed processes that are "locally consistent", *i.e.*, that are built out of sequences of primitive actions assumed to preserve the consistency of the system. We make this intuition clearer in the following.

To define consistency without talking about the primitive operations on the memory, we assume that we are given a set of finite sequences of primitive actions. The elements of this set will be called *locally consistent sequences*. Intuitively, a locally consistent sequence is intended to preserve the consistency of the global system if executed completely and in order, but will not necessarily do so if it is executed partially or with the interleaving of other primitive actions. So with respect to a given set of locally consistent sequences, a *locally consistent process* is a well-formed process in which every occurrence of a primitive action must be part of a locally consistent sequence inside atomic brackets.

3 Equational theory

We now define an equivalence relation on processes that is meant to reflect our informal understanding of all-or-nothing actions, isolation, durability, concurrency and failure. This equivalence relation will be defined as the smallest congruence that contains a set of equality axioms. We are thus defining an equational theory.

Structural equalities. The first set of equality axioms are called *structural equalities* because they reflect obvious facts about the algebraic structure of the composition operators and skip.

– Parallel composition (||) is associative (1), commutative (2) and has skip as identity element (3).
– Sequential composition (;) is associative (4) and has skip as right identity (5) and left identity (6).
– Alternation (+) is associative (7), commutative (8) and idempotent (9).
– Furthermore, alternation distributes over every other operator, capturing the idea that a choice made at a deep level of the program determines a choice for the entire program.

$$(P + Q)\ ;\ R = P\ ;\ R + Q\ ;\ R \tag{10}$$
$$P\ ;\ (Q + R) = P\ ;\ Q + P\ ;\ R \tag{11}$$
$$(P + Q)\ ||\ R = P\ ||\ R + Q\ ||\ R \tag{12}$$
$$\langle P + Q\rangle_k = \langle P\rangle_k + \langle Q\rangle_k \qquad k \in \{A, D, I\} \tag{13}$$
$$\mathit{crash}(P + Q) = \mathit{crash}(P) + \mathit{crash}(Q) \tag{14}$$

– An empty action has no effect.

$$\langle \mathsf{skip}\rangle_k = \mathsf{skip} \qquad k \in \{A, D, I\} \tag{15}$$

Interleaving equality. Isolation means that a parallel composition of two isolated processes must be equivalent to some sequential composition of these processes. This corresponds to the following *interleaving law*:

$$\begin{aligned}\langle P\rangle_I\ ;\ P'\ ||\ \langle Q\rangle_I\ ;\ Q' = {} & \langle P\rangle_I\ ;\ (P'\ ||\ \langle Q\rangle_I\ ;\ Q') + \\ & \langle Q\rangle_I\ ;\ (Q'\ ||\ \langle P\rangle_I\ ;\ P')\end{aligned} \tag{16}$$

Global failure equalities. When we write $\mathit{crash}(P)$ we indicate that one or more failures, each followed by a recovery, will happen during the execution of P. The equalities below make it possible to find equivalent processes that are simpler in the sense that we know more accurately where failures can possibly take place.

– If failures occurred during the sequence $P\ ;\ Q$ they might have occurred during P, or during Q, or during both P and Q:

$$\mathit{crash}(P\ ;\ Q) = \mathit{crash}(P)\ ;\ \mathit{crash}(Q) \tag{17}$$

According to our intuitive understanding of $\mathit{crash}(P)$, namely, one or more failures during P, equation (17) is not very natural. It seems that we might expect to have two or more failures on the right-hand side, whereas there is only one or more on the left-hand side. Maybe it would have been more natural to write

$$\mathit{crash}(P\ ;\ Q) = P\ ;\ \mathit{crash}(Q) + \mathit{crash}(P)\ ;\ Q + \mathit{crash}(P)\ ;\ \mathit{crash}(Q)$$

In fact, the above equality holds in our theory, because $P\ ;\ \mathit{crash}(Q)$ and $\mathit{crash}(P)\ ;\ Q$ can be shown to be "special cases" of $\mathit{crash}(P)\ ;\ \mathit{crash}(Q)$. (We say that Q is a "special case" of P if there exists a process R such that $P = Q + R$.)

– Based on our informal definition of the operator *crash*() it is natural to see every process $\mathit{crash}(P)$ as a fixed point of *crash*(): contaminating an already-failed process with additional failures has no effect.

$$\mathit{crash}(\mathit{crash}(P)) = \mathit{crash}(P) \tag{18}$$

– A crashed primitive action may either have executed normally or not have executed at all. But in each alternative we propagate the failure to the left so that it can affect already performed actions.

$$\mathit{crash}(a) = \downarrow\uparrow \ ; a + \ \downarrow\uparrow \tag{19}$$

– A crashed A-action behaves in the same way, in accordance with its informal "all or nothing" meaning.

$$\mathit{crash}(\langle P \rangle_A) = \downarrow\uparrow \ ; \langle P \rangle_A + \ \downarrow\uparrow \tag{20}$$

– A failure during a D-action is propagated both inside the D-brackets so that it can affect the nested process, and before the D-brackets so that it can affect previously performed actions.

$$\mathit{crash}(\langle P \rangle_D) = \downarrow\uparrow \ ; \langle \mathit{crash}(P) \rangle_D \tag{21}$$

– Since we consider only well-formed processes, we know that a term of the form $\mathit{crash}(P)$ cannot be composed in parallel with any other term. Hence, isolation brackets directly inside $\mathit{crash}(\cdot)$ are superfluous.

$$\mathit{crash}(\langle P \rangle_I) = \mathit{crash}(P) \tag{22}$$

Failure event equalities. We will now consider the effect of a failure event ($\downarrow\uparrow$) on actions that have already been performed.

– If the failure was preceded by a primitive action or an A-action, then either the effect of those actions is completely undone, or the failure did not have any effect at all. In either case we propagate the failure event to the left so that it can affect previous actions.

$$a \ ; \ \downarrow\uparrow = \downarrow\uparrow \ ; a + \ \downarrow\uparrow \tag{23}$$

$$\langle P \rangle_A \ ; \ \downarrow\uparrow = \downarrow\uparrow \ ; \langle P \rangle_A + \ \downarrow\uparrow \tag{24}$$

– A crash/recovery can in principle act on every action that precedes it. But a durable transaction that has completed becomes, by design, resistant to failure.

$$\langle P \rangle_D \ ; \ \downarrow\uparrow = \downarrow\uparrow \ ; \langle P \rangle_D \tag{25}$$

– If an I-action is followed by a failure event, we know that parallel composition is impossible, so the isolation brackets are again superfluous.

$$\langle P \rangle_I \ ; \ \downarrow\uparrow = P \ ; \ \downarrow\uparrow \tag{26}$$

Nested failure equalities. The effects of a failure are local. This means that a failure inside some action cannot escape it to affect outer actions. Furthermore, a crash/recovery at the beginning of an action has no effect on the action's state, because nothing has been done yet. We can safely ignore such crash/recoveries if the enclosing action is isolated or durable:

$$\langle \downarrow\uparrow \ ; P \rangle_D = \langle P \rangle_D \tag{27}$$

$$\langle \downarrow\uparrow \ ; P \rangle_I = \langle P \rangle_I \tag{28}$$

By contrast, a crash/recovery at the beginning of an atomic action will abort that action:

$$\langle \downarrow\uparrow \ ; P \rangle_A = \mathsf{skip} \tag{29}$$

This is realistic, since a failure that occurs after the start of an atomic action will have the effect of aborting that action, no matter whether the action has already executed some of its internal code or not. This is also necessary from a technical point of view, since a crash/recovery at the beginning of an atomic action might be the result of rewriting a later crash/recovery using laws (19), (20), (23) or (24). In that case, the sequence of sub-actions inside the $\langle \cdot \rangle_A$ may be only partial and therefore must be dropped in order to satisfy the all-or-nothing principle.

Admissible Equalities. Using the equational theory, we can show that the following four equalities also hold.

$$\downarrow\uparrow \ ; \ \downarrow\uparrow = \downarrow\uparrow$$

$$\mathit{crash}(\downarrow\uparrow) = \downarrow\uparrow$$

$$\mathit{crash}(P) \ ; \ \downarrow\uparrow = \mathit{crash}(P)$$

$$\downarrow\uparrow \ ; \mathit{crash}(P) = \mathit{crash}(P)$$

The first two equalities are simple consequences of laws (6), (17) and (18).

The last two equalities are not directly derivable from the given axioms. However, one can show that they hold for all processes that result from substituting a concrete well-formed closed process for the meta-variable P.

The "harmless case" property. Among the possible effects of a failure there is always in our system the case where the crash/recovery has no effect. More precisely, for every process P, it holds that $\downarrow\uparrow \ ; P$ is a special case of $\mathit{crash}(P)$. This conforms to the usual intuition of failure: we know that something has gone wrong but we do not know exactly what the effects have been, so we must also consider the case where nothing bad occurred.

4 Meta-theoretical properties

We now establish the two main theorems of our calculus. The first is that every process is equivalent to a unique process in canonical form. Since canonical forms contain neither parallel compositions nor failures (except at the very beginning), this gives us a theory for reasoning about decomposed transactions with failures.

Theorem 1 (Existence and Uniqueness of Canonical Form). *For each well-formed process P there is one equivalent process in canonical form. Furthermore this later process is unique modulo associativity, commutativity and idempotence of $(+)$ (axioms (7), (8), (9)), associativity of $(;)$ (axiom (4)) and the simplifications involving* skip *(axioms (5), (6), (15)). We call this process the* canonical form of P.

The proof of this theorem is based on a rewriting system modulo a set of structural rules which is shown equivalent to the equational theory. Canonical forms are then normal forms (irreducible terms) of the rewriting system.

The second theorem is that the reduction of a process to its canonical form preserves its consistency. This theorem guarantees that our equational calculus conforms to the usual behavior of a transaction system, which requires that local consistency of transactions implies global consistency of the system.

Theorem 2 (Preservation of Consistency). *The canonical form of a locally consistent process is also a locally consistent process.*

Conclusion

This paper presents an axiomatic, equational semantics for all-or-nothing, isolated, and durable actions. Such actions may be nested, and may be composed using parallel composition, sequential composition and alternation. Traditional transactions correspond to nested A-D-I-actions. The semantics is complete, in the sense that it can be used to prove that local consistency of individual transactions implies global consistency of a transaction system.

The work done in this paper could be used to better understand the interplay between actions that guarantee only some of the ACID properties. These kinds of actions are becoming ever more important in application servers and distributed transaction systems, which go beyond centralized databases.

We have argued informally that our axioms capture the essence of decomposed transactions. It would be useful to make this argument more formal, for instance by giving an abstract machine that implements a transactional store, and then proving that the machine satisfies all the equational axioms. This is left for future work.

Another continuation of this work would consider failures that can escape their scope and affect enclosing actions in a limited way. A failure could then be tagged with an integer, as in $\downarrow\uparrow_n$, which would represent its *severity*, *i.e.*, the number of failure-tight compartments that the failure can go through.

The process notion presented in this paper is a very high-level abstraction of a transaction system. This is a first attempt at formalizing transactions, and has allowed us to prove some properties which are only true at this level of abstraction. Now it is necessary to refine our abstraction in order to take into account replication, communication and distribution. We hope to find a *conservative* refinement, *i.e.*, a refinement for which the proofs in this paper still work.

References

1. David. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Proceedings of the ACM Conference on Language Design for Reliable Software*, volume 12 of *SIGPLAN Notices*, pages 128–137, 1977.
2. Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *COMPSUR*, 15(4):287–317, 1983.
3. J.E.B Moss. Nested transactions: An approach to reliable distributed computing. MIT Press, March 1985.
4. Philip Bernstein, Vassos Hadzilacos, and Nathan Goodman. Concurrency Control and Recovery in Database Systems. Addison-Wesley. 1987.
5. Andrew P. Black. Understanding transactions in the operating system context. *Record of the Fourth ACM SIGOPS European Workshop, Operating Systems Review*, 25(1):73–76, 1991. Workshop held in Bologna, Italy, September 1990.
6. J. Gray and A. Reuter. Transaction Processing: Techniques and Concepts. Morgan Kaufman. 1992
7. A.Z Spector et al. Support for distributed transactions in the TABS prototype. IEEE Transactions on Software Engineering, 11 (6). June 1985.
8. J. Eppinger, L. Mummert, and A. Spector. Camelot and Avalon: A Distributed Transaction Facility. Morgan Kaufmann Publishers, 1991.
9. B. Liskov and R. Scheifler. Guardians and actions: Linguistic support distributed programs. ACM Transactions on Programming Languages and Systems, July 1993.
10. G. Parrington and S. Schrivastava. Implementing concurrency control in reliable distributed object-oriented systems. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'88)*, *LNCS*, August 1988. Norway.
11. D. Detlefs, M. Herlihy and J. Wing. Inheritance of synchronization and recovery properties in Avalon/C++. IEEE Computer, December 1988.
12. R. Guerraoui, R. Capobianchi, A. Lanusse and P. Roux. Nesting Actions through Asynchronous Message Passing: the ACS protocol. Proceedings of the European Conference on Object-Oriented Programming, Springer Verlag, LNCS, 1992.
13. J. Wing. Decomposing and Recomposing Transactional Concepts. Object-Based Distributed Programming. R. Guerraoui, O. Nierstrasz and M.Riveill (eds). Springer Verlag (LNCS 791).
14. P. Chrysantis and K. Ramamritham. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. ACM SIGMOD International Conference on Management of Data. 1990.
15. http://www.arjuna.com/products/arjunats/features.html
16. http://www.bea.com/framework.jsp?CNT=homepage_main.jsp&FP=/content
17. http://www-3.ibm.com/software/info1/websphere/index.jsp
18. http://www.microsoft.com/com/tech/MTS.asp
19. http://java.sun.com/products/ejb/
20. Martin Berger and Kohei Honda. The Two-phase Commitment Protocol in an Extended Pi-calculus. In Proceedings of EXPRESS '00, ENTCS, 2000.
21. A. Bernstein, P. Lewis and S. Lu. Semantic Conditions for Correctness at Different Isolation Levels 16the Int'l Conf. on Data Engineering, 2000.
22. Andrew P. Black, Vincent Cremet, Rachid Guerraoui, Martin Odersky. An Equational Theory for Transactions. EPFL Technical Report IC/2003/26, 2003.