PEPITO

IST-2001-33234

PEer-to-Peer Implementation and TheOry

Deliverable no: D1.3

Axiomatisation of Transactions

REPORT VERSION: first

REPORT PREPARATION DATE: 2003.06.18

CLASSIFICATION: Public

DELIVERABLE NO: D1.3 DUE DATE: Month 18 DELIVERY DATE: Month 18

PROJECT START DATE: 2002.01.01 PROJECT DURATION: 36 months

RESPONSIBLE PARTNER: EPFL

PARTICIPATING PARTNERS: EPFL

PROJECT COORDINATOR: Swedish Institute of Computer Science AB

PROJECT PARTNERS: EPFL Lausanne, INRIA Paris, KTH Stockholm, UCL Louvain, University of Cambridge UK

Project funded by the European Community under the 'Information Society Technologies' Programme (1998–2002)

Project Number: IST-2001-33234
Project Acronym: PEPITO
Title: PEer-to-Peer Implementation and TheOry
Deliverable No: D1.3
Axiomatisation of Transactions
Due date: project month 18
Delivery date: 2003-06-18

Responsible Partner: EPFL
Participating Partners: EPFL

18th June 2003

Prepared by Vincent Cremet and Martin Odersky, with input from Rachid Guerraoui and Andrew Black

Contents

Overview

The work presented in this document is part of the workpackage 1 of the PEPITO project. This workpackage aims at defining formal models for aspects of distributed computation which are central for peer-to-peer systems. Transactions fall into this category and can help to design robust and fault-tolerant distributed programs. This final report contains our results concerning the axiomatization of the properties of transactions.

Transactions are commonly described as being Atomic, Consistent, Isolated and Durable: the ACID properties. However, although these words convey a powerful intuition, the properties of atomicity, consistency, isolation and durability have never been given a precise semantics, in a way that disentangles each property from the others but allows their composition. 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 document provides 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 global consistency of the whole system from the local consistency of the individual actions. The three kinds of action can be nested, leading to different forms of transactions, each with a well-defined semantics. Conventional transactions are obtained as ADI-actions.

From the equational semantics we develop a formal proof principle for transactional programs. To show that a property holds for all transactional programs, it suffices to demonstrate the property for all programs in so-called "canonical form", a form that contains neither parallel composition nor embedded failures. Using this principle, we derive the induction rule for Consistency.

We hope that this theoretical work on the decomposition of transactions can be applied to the design and implementation of simple and well-understood language constructs for transactional computation in a peer-to-peer context. Contrary to the domain of databases where a transaction is always supposed to satisfy the four ACID properties, using transactions at the level of applications can lead to choose only a subset of these properties to match specific requirements, for instance in terms of efficiency.

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.

1.1 Transactions

The most successful abstraction for coping with failure is the transaction, which has emerged from earlier notions of atomic action [11]. The most popular characterization of transactions is due to

Haerder and Reuter [10], 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 that 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 idea formally (see theorem 10.2).

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 and distributed transactions. 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.

1.2 Background

Transactions promise the programmer that he will never see a failure. In a 1991 paper [1], 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

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

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

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 5). Thus, the main contribution of the 1991 paper was the identification of an interesting problem, not its solution. That was not trivial and 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.

1.3 Formalizing Failure

To be more precise about what a failure really is, we assume that computation involves a set of stateful objects that are manipulated by a set of actions. We are thus concerned with two kinds of failure: failure of objects and failure of actions.

Objects can fail, with the consequence that they "loose" their state. Recovery will restore the failed object to some prior state, but we cannot in general assume that the restored state will be the same as the state immediately before the failure. Informally, we might imagine that the object is restored to the last state that was "saved" by being durably written to persistent storage.

More formally, we consider the state as a pair of functions S_D, S_V : objects $\rightarrow$ values, where S_D represents the durable state and S_V the volatile state. A failure followed by a recovery results in S_V being reset to S_D. More generally, we can represent the state as a sequence of functions $S_0, S_1, ..., S_n$: objects $\rightarrow$ values, where S_0 is the most durable state, and the S_i are progressively more volatile as i increases.

Actions can fail, which means that they spontaneously *abort*. The notion of an aborted action is similar to that captured by the abort statement in Dijkstra's language of guarded commands [4]. Specifically, no assertion can be made about the state of an aborted computation: the computation no longer exists. In this setting, transactions can be viewed as a mechanism that bounds *how much* computation is aborted when an action fails.

Relationship between the two kinds of failures. Failure of objects make reasoning about the behavior of distributed programs very difficult. Consider an object x that does no more than preserve a value written to its field. If x is local to a program that performs an action on x, it is reasonable to assume that x respects the axiom of assignment. Thus, after setting x's field to 1, say by sending the (local) message $x.set(1)$, we expect that field to be 1, that is, we expect the assertion $x.get() = 1$ to hold. The assumption of locality means that if x fails, the action that manipulates x also fails, and thus we are deprived of the opportunity to observe $x.get()$.

Now suppose that x is manipulated by an action executing on a remote machine m, so x and m can fail independently. Now we cannot in general assume that after sending the message $x.set(1)$, the assertion $x.get() = 1$ holds. This is because the object x may have failed and subsequently been recovered in a prior state in which $x.get() = 42$. However, because m (and the action executing on machine m) may *not* have failed, we can now observe $x.get()$. In other words, in the presence of partial failure, even the most basic rules for reasoning about computations do not hold.

We avoid this problem by linking the failure of an object to the failure of the actions that manipulate it. If an object x fails, we cause all of the innermost uncommitted actions from which x is reachable to fail also. That is, any action that might operate on x aborts. Although this sounds rather draconian, it is in fact far less so than what happens in real transaction processing systems. In real systems, the failure of a data object may cause *all* executing actions to abort. This may be much simpler than trying to ascertain whether a given object is reachable from a particular action.

1.4 Nested failures

Consider an interior action I nested inside an enclosing action E. Suppose that I manipulates and depends on the object x (see (3)). Based on the above discussion, if x fails, then I will abort. What about E?

$$\overbrace{\langle ... \underbrace{\langle x.set(1) \;;\; ... \;;\; x.get() \;;\; ... \rangle}_{I} \;;\; ... \rangle}^{E} \tag{3}$$

There are two cases to consider. In the first case, E also manipulates x directly, and so E will also abort. In this case it makes no sense to ask about the state of E. In the second case, E does not depend on x, so it is reasonable to expect that E will recognize that I has aborted, and somehow continue its own work.

2 Related work

Transactions originated in the database arena [16] and were eventually ported to distributed operating-systems like Tabs [20] or Camelot [15], as well as to distributed programming languages like Argus [17], Arjuna [12], Avalon [14] or Venari [21]. 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.

For instance, the Camelot system provided the abstraction of “recoverable virtual memory”, which supported two kinds of transactions: all-or-nothing transactions (called “no-flush” transactions), and all-or-nothing transactions that are also durable [15]. Concurrency-control was factored out into a separate mechanism that the programmer could use to ensure isolation. This led to a very flexible way to trade transaction properties with efficient implementations, but their meaning was not 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. The interpretation of any combination of those was again 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 [13] 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 [9], BEA Weblogics [7], IBM Webspheres [6], Micosoft MTS [8], and Sun EJB [5] 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.

3 The Equational semantics

Our equational theory does not deal with objects directly. Instead we assume the existence of a set of primitive atomic operations a, b, c that may access state, and which may fail, either because the state on which they rely fails, or because the machine on which they are executing crashes. We do not distinguish these cases: $\mathit{crash}(a)$ means that one or more failures and subsequent recoveries took place during the execution of a.

We distinguish three kinds of actions: A-actions (which capture the all-or-nothing property), D-actions (which capture the durability property), and I-actions (which capture the isolation property).

The 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.

By turning the axioms into a rewriting system modulo some structural equalities, we prove that every process has a unique *canonical form* (Theorem 10.1). 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 10.2). Hence, we can show that in our model of transactions, local consistency implies global consistency.

4 Pre-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.

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

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.

In order to not limit ourselves to a particular kind of system, we leave the nature of the primitive actions completely unspecified. Consequently, we must treat them as atomic because we cannot say anything about the partial effect of a 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 always has the 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 1.1, 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.

As discussed in section 1.3, 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 local to the memory objects accessible from that action. For instance in the action $\langle\langle P \rangle_I \mid\mid \langle Q \; ; \; \downarrow\uparrow\rangle_I\rangle_A$ 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.

The identification of action brackets with failure scopes is attractive because it simplifies the equational calculus. But there is no fundamental reason why the two must be the same. It would be quite possible to introduce a different scoping operator for failures, at the price of some additional complexity in our calculus.

Process composition.

Processes can be combined in three different ways. Sequential composition (;) expresses that one process is executed before another. Parallel composition (||) expresses two processes which execute concurrently. Alternation (+) expresses a non-deterministic choice between two processes.

Null action.

The null action is represented by the symbol skip. For instance, $P + \mathsf{skip}$ means "either do P or do nothing". skip is an identity of both sequential and parallel composition.

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 a shorthand notation to represent such a single failure event:

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

One might wonder why we did not instead start by taking the symbol $\downarrow\uparrow$ as a primitive and letting $\mathit{crash}(P)$ be an abbreviation for $\downarrow\uparrow \,||\, P$. This was in fact our initial approach, but we encountered problems that led us to realize 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 5.

The motivation for dealing with crash and recovery as an inseparable pair of events is that, while the effect of a crash might be to take a memory object to an arbitrary state, we can assume that the associated recovery will restore it to a state that might have been obtained from some sequence of primitive actions. That is, if a crash takes place during or following a sequence of primitive actions, we are assured that recovery will establish a state that could have arisen from the execution of some subset of these actions (not necessarily a prefix).

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.

Finally, note that we never know if a crash/recovery will effectively damage some information or not, so in all the equations dealing with failures there is always one alternative that leaves the crashed process untouched.

5 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. For instance:

- ▷ $\langle a_1 \ ; \ b_1\rangle_I \ || \ \langle a_2 \ ; \ b_2\rangle_I$ can produce only $a_1 \ ; \ b_1 \ ; \ a_2 \ ; \ b_2$ or $a_2 \ ; \ b_2 \ ; \ a_1 \ ; \ b_1$, whereas $(\langle a_1\rangle_I \ ; \ \langle b_1\rangle_I) \ || \ (\langle a_2\rangle_I \ ; \ \langle b_2\rangle_I)$ can produce any permutation of the actions a_1, b_1, a_2 and b_2 where a_1 is before b_1 and a_2 before b_2.

- ▷ In contrast, $\langle P\rangle_D \ ; \ \langle Q\rangle_A$ is an example of a non-interleavable process because P and Q are not enclosed in isolation brackets. Hence, the effect of the parallel composition of P or Q with some other process is not well-defined.

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 only allowed for interleavable processes.

$$
\begin{array}{llcll}
\textit{Process} & P, Q, R & ::= & a, b, c & \\
 & & | & \langle P\rangle_k & k \in \{A, D, I\} \\
 & & | & I \ || \ J & \\
 & & | & P \ ; Q & \\
 & & | & P + Q & \\
 & & | & \mathsf{skip} & \\
 & & | & \textit{crash}(P) & \\
 & & & & \\
\textit{Interleavable process} & I, J & ::= & \langle P\rangle_I & \\
 & & | & I \ || \ J & \\
 & & | & I \ ; \ J & \\
 & & | & I + J & \\
 & & | & \mathsf{skip} &
\end{array}
$$

An important property of well-formed processes is that $\textit{crash}(P)$ 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, $\langle P\rangle_I \ || \ \langle Q\rangle_I$ 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!

6 Canonical processes

A process is in *canonical form* if it belongs to the following syntactic class.

$$
\begin{array}{llcll}
\textit{Canonical process} & S & ::= & S' & \\
 & & \mid & \downarrow\uparrow\ ; S' & \\
 & & \mid & S + S & \\
 & S' & ::= & a & \\
 & & \mid & \mathsf{skip} & \\
 & & \mid & \langle S' \rangle_k & k \in \{A, D, I\} \\
 & & \mid & S' \,;\, S' &
\end{array}
$$

Informally, a process in canonical form consists of a non-deterministic 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.

We will show later that for each process P there exists an equivalent process in canonical form, and that the canonical process is unique modulo the obvious structural equivalences involving alternation, sequential composition and skip (see Theorem 10.1 for more details). We will call the latter process *the canonical form of* P.

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 to an equivalent process C in canonical form, and show 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.

7 Locally consistent processes

We now define a class of well-formed processes that are "locally consistent", *i.e.*, which are built from sequences of primitive actions that are 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 transaction brackets.

To illustrate this notion of consistency, consider a system with a single integer variable n and two primitive actions, double and $\mathsf{sub2}$, which respectively double and subtract 2 from n. The only consistent state of the system is when $n = 2$; this is also the initial state. Now the locally consistent sequences in this system will be all sequences of double's and $\mathsf{sub2}$'s such that the system state after applying all the elements of the sequence is again consistent. For instance, $\mathsf{double}\ ;\ \mathsf{sub2}$ is locally consistent, as is $\mathsf{double}\ ;\ \mathsf{double}\ ;\ \mathsf{sub2}\ ;\ \mathsf{sub2}\ ;\ \mathsf{sub2}$. Note how concurrency affects consistency. If two instances of the locally consistent sequence $\mathsf{double}\ ;\ \mathsf{sub2}$ execute in parallel, the interleaving $\mathsf{double}\ ;\ \mathsf{sub2}\ ;\ \mathsf{double}\ ;\ \mathsf{sub2}$ preserves consistency but the interleaving $\mathsf{double}\ ;\ \mathsf{double}\ ;\ \mathsf{sub2}\ ;\ \mathsf{sub2}$ does not. Similarly, a failure that prevents a sequence from completing or that undoes a primitive action might cause the state to become inconsistent.

Given this notion of local consistency, we can state a theorem of "preservation of consistency". Theorem 10.2 says that the canonical form of a locally consistent process is itself locally consistent. With this theorem and the proof principle explained in the previous section, we precisely capture the intuition behind the consistency guarantee of a transaction system: that local consistency of individual transactions implies global consistency of the transaction system.

8 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, commutative and has skip as identity element.

$$(P \mid\mid Q) \mid\mid R = P \mid\mid (Q \mid\mid R) \tag{1}$$
$$P \mid\mid Q = Q \mid\mid P \tag{2}$$
$$P \mid\mid \mathsf{skip} = P \tag{3}$$

▹ Sequential composition is associative and has skip as identity element.

$$(P\ ;\ Q)\ ;\ R = P\ ;\ (Q\ ;\ R) \tag{4}$$
$$P\ ;\ \mathsf{skip} = P \tag{5}$$
$$\mathsf{skip}\ ;\ P = P \tag{6}$$

▹ Alternation is associative, commutative and idempotent.

$$(P + Q) + R = P + (Q + R) \tag{7}$$
$$P + Q = Q + P \tag{8}$$
$$P + P = P \tag{9}$$

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.

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

▹ 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*:

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

Global failure equalities

When we write *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 the 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:

  $$\textit{crash}(P\,;Q) \quad = \quad \textit{crash}(P)\,;\textit{crash}(Q) \tag{17}$$

  According to our intuitive understanding of *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

  $$\begin{aligned}\textit{crash}(P\,;Q) \quad &= \quad P\,;\textit{crash}(Q) + \textit{crash}(P)\,;Q \\ & + \textit{crash}(P)\,;\textit{crash}(Q)\end{aligned}$$

  In fact, the above equality holds in our theory, because $P\,;\textit{crash}(Q)$ and $\textit{crash}(P)\,;Q$ are in fact "special cases" of $\textit{crash}(P)\,;\textit{crash}(Q)$. (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 *crash*(P) as a fixed point of *crash*$()$: contaminating an already-failed process with additional failures has no effect.

  $$\textit{crash}(\textit{crash}(P)) \quad = \quad \textit{crash}(P) \tag{18}$$

- 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.

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

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

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

▷ 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.

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

▷ 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 inside $\mathit{crash}(\cdot)$ are superfluous.

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

Failure event equalities

We will now consider the effect of a failure event 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.

$$a \; ; \; \downarrow\uparrow \quad = \quad \downarrow\uparrow \; ; a + \downarrow\uparrow \tag{23}$$
$$\langle P \rangle_A \; ; \; \downarrow\uparrow \quad = \quad \downarrow\uparrow \; ; \langle P \rangle_A + \downarrow\uparrow \tag{24}$$

▷ 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.

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

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

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

Nested failure equalities

The effects of a failures 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:

$$\langle \downarrow\uparrow \; ; P \rangle_D \quad = \quad \langle P \rangle_D \tag{27}$$
$$\langle \downarrow\uparrow \; ; P \rangle_I \quad = \quad \langle P \rangle_I \tag{28}$$

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

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

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. It 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 $\langle\cdot\rangle_A$ is only partial and therefore must be dropped in order to satisfy the all-or-nothing principle.

For example consider the process $\langle a\ ;\ \ \downarrow\uparrow\ \ ; P\rangle_A$. This process is, by (23), equivalent to $\langle \downarrow\uparrow\ ; P\rangle_A + \langle \downarrow\uparrow\ ; a\ ; P\rangle_A$. If a crash/recovery at the beginning of an A-action did not necessarily abort that action, then $\langle P\rangle_A$ would be a possible outcome of $\langle \downarrow\uparrow\ ; P\rangle_A$. This means that only part of the original A-action would have been executed, whereas we require "all-or-nothing".

9 Properties

In this section, we establish a number of properties that follow from the equational theory.

Admissible Equalities

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

$$\begin{array}{rcl} \downarrow\uparrow\ ;\ \downarrow\uparrow & = & \downarrow\uparrow \\ \mathit{crash}(\downarrow\uparrow) & = & \downarrow\uparrow \\ \mathit{crash}(P)\ ;\ \downarrow\uparrow & = & \mathit{crash}(P) \\ \downarrow\uparrow\ ; \mathit{crash}(P) & = & \mathit{crash}(P) \end{array}$$

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

$$\begin{array}{rcll} \downarrow\uparrow\ ;\ \downarrow\uparrow & \equiv & \mathit{crash}(\mathsf{skip})\ ; \mathit{crash}(\mathsf{skip}) & \\ & = & \mathit{crash}(\mathsf{skip}\ ;\ \mathsf{skip}) & \text{by (17)} \\ & = & \mathit{crash}(\mathsf{skip}) & \text{by (6)} \\ & \equiv & \downarrow\uparrow & \end{array}$$

And:

$$\begin{array}{rcll} \mathit{crash}(\downarrow\uparrow) & \equiv & \mathit{crash}(\mathit{crash}(\mathsf{skip})) & \\ & = & \mathit{crash}(\mathsf{skip}) & \text{by (18)} \\ & \equiv & \downarrow\uparrow & \end{array}$$

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 all process P, $\downarrow\uparrow\ ; P$ is a special case of *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.

Failure during parallel computation

In this section we demonstrate the expressive power of our calculus by studying the example of a failure during a parallel computation.

Note that there is no rule for reducing terms of the form $\mathit{crash}(P \mid\mid Q)$. This is not an oversight. In fact, in a previous version of the calculus we tried treating a failure as a process that could be put in parallel with other processes. We discovered that this leads to an intrinsically non-confluent system, and realized that treating a failure as a special kind of interleavable process makes no sense. So we abandoned this idea and instead introduced the operator $\mathit{crash}(\cdot)$.

We now study in detail the behavior of two concurrent processes that can fail. We can derive:

$$\begin{array}{rcll} \mathit{crash}(\langle P\rangle_I \mid\mid \langle Q\rangle_I) & = & \mathit{crash}(\langle P\rangle_I \,;\, \langle Q\rangle_I + \langle Q\rangle_I \,;\, \langle P\rangle_I) & \text{by (16)} \\ & = & \mathit{crash}(\langle P\rangle_I \,;\, \langle Q\rangle_I) + \mathit{crash}(\langle Q\rangle_I \,;\, \langle P\rangle_I) & \text{by (14)} \\ & = & \mathit{crash}(\langle P\rangle_I) \,;\, \mathit{crash}(\langle Q\rangle_I) + \mathit{crash}(\langle Q\rangle_I) \,;\, \mathit{crash}(\langle P\rangle_I) & \text{by (17)} \end{array}$$

Rule (17) might seem surprising since the crash on the left-hand side appears to be duplicated in the operands on the right-hand side. To justify the rule, observe that real implementation of transactions can execute parallel actions concurrently, even if they were meant to be isolated. For example, this can happen if the sets of resources accessed by the transactions are disjoint. One must ensure only that the observable effect of each run is the same as the effect of some serialization of the I-actions.

Without loss of generality, assume that P appears to commit before Q starts. We have to take into account three main cases: either the crash/recovery occurs during the overlapping of the transactions, or it occurs during P (before Q starts to execute), or it occurs during Q after P is finished.

In the first case, both transactions must have been affected by the crash, but to an external observer transaction P must have appeared to execute completely before Q started. So such an observer should see something like $\mathit{crash}(\langle P\rangle_I) \,;\, \mathit{crash}(\langle Q\rangle_I)$, that is, the observer should see two crashes. In the two other cases, the crash/recovery occurred during one of the transactions, so we should observe either $\mathit{crash}(\langle P\rangle_I) \,;\, \langle Q\rangle_I$ or $\langle P\rangle_I \,;\, \mathit{crash}(\langle Q\rangle_I)$.

However, in our theory $\mathit{crash}(P) \,;\, Q$ and $P \,;\, \mathit{crash}(Q)$ are special cases of the process $\mathit{crash}(P) \,;\, \mathit{crash}(Q)$ for all P and Q. This follows from the "harmless case" property and from some of the admissible equalities. Let us show for instance that $\mathit{crash}(P) \,;\, Q$ is a special case of $\mathit{crash}(P) \,;\, \mathit{crash}(Q)$:

$$\begin{array}{rcll} \mathit{crash}(P) \,;\, \mathit{crash}(Q) & = & \mathit{crash}(P) \,;\, (\downarrow\uparrow \,;\, Q + R) & \text{“harmless case” property} \\ & = & (\mathit{crash}(P) \,;\, \downarrow\uparrow) \,;\, Q + \mathit{crash}(P) \,;\, R & \text{by (11) then (4)} \\ & = & \mathit{crash}(P) \,;\, Q + \mathit{crash}(P) \,;\, R & \text{third admissible law} \end{array}$$

From actions to transactions

The various kinds of actions are not commutative. For instance: $\langle\langle P\rangle_I\rangle_A = \langle\langle P\rangle_A\rangle_I$ does not hold, because $\langle\langle P\rangle_I\rangle_A$ cannot be interleaved, whereas $\langle\langle P\rangle_A\rangle_I$ can. The same remark holds for $\langle\rangle_I$ with $\langle\rangle_D$.

Neither does $\langle\cdot\rangle_A$ commute with $\langle\cdot\rangle_D$. The intuition behind the difference is that an A-action can be undone by a subsequent failure, whereas a D-action cannot — that is the meaning of durable.

Formally, we have the equivalences

$$\begin{array}{lcll} \langle\langle a\rangle_D\rangle_A \; ; \; \downarrow\uparrow & = & \downarrow\uparrow \; ; \langle\langle a\rangle_D\rangle_A + \downarrow\uparrow & \text{by (24)} \\ \langle\langle a\rangle_A\rangle_D \; ; \; \downarrow\uparrow & = & \downarrow\uparrow \; ; \langle\langle a\rangle_A\rangle_D & \text{by (25)} \end{array}$$

but the right-hand sides are observably different: the latter must execute the primitive action a whereas the former may also do nothing. The right-hand sides are also inequivalent in our equational theory: both are canonical forms, and as we are about to show (see Theorem 10.1), each process has a unique canonical form.

Thus, what we could call a *complete* transaction should be of the form $\langle\langle\langle P\rangle_A\rangle_D\rangle_I$. Nested in this way it is (1) interleavable, because the isolation bracket is outermost, (2) it is protected from crash as soon as it completes, because the next outermost bracket is a durability bracket, and (3) it executes completely or not at all. In general, it is the charge of the designer to put the brackets in the order that will make the program behave as required.

10 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.

The second 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 behaviour of a transaction system, which requires that local consistency of transactions implies global consistency of the system.

Theorem 10.1 (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.

Proof sketch. In order to prove that there is exactly one process in canonical form in each equivalence class, it is easier to work with a rewriting system whose reflexive, symmetric, transitive closure coincides with the equational theory. Such a rewriting system is presented in appendix A. Processes that cannot be reduced by the action of the rewriting system are said to be in *normal form*.

The major steps of the proof are then to prove the following lemmas.

1. The reflexive, symmetric, transitive closure of the rewriting relation coincides with the equational equivalence. More formally, for all processes P and Q

$$P = Q \qquad \text{iff} \qquad P \leftrightsquigarrow^* Q.$$

2. The rewriting system is terminating, *i.e.*, there is no infinite reduction sequence.

3. The rewriting system is confluent (because it is terminating and locally confluent) for well-formed processes, *i.e.*, two processes that are the reduced forms of a common process have also a common reduct.

4. Reduction preserves well-formedness.

5. A well-formed process in normal form is in canonical form.

6. A process in canonical form can be reduced to a process in normal form using only the rules of associativity of $(;)$ idempotence of $(+)$ and simplifications involving skip (when skip is alone in an action or is part of a sequence). We write these reductions $A(;)$, $I(+)$ and $S(\mathsf{skip})$.

The entire proof can then be summarized in the following diagram:

$$\begin{tikzcd} P \arrow[r, phantom, "="] \arrow[d, squiggly, "*"'] & Q \arrow[d, squiggly, "*"', "A(;)\ I(+)\ S(\mathsf{skip})"] \\ P_{\downarrow} \arrow[r, phantom, "="' , "="] & Q_{\downarrow} \end{tikzcd}$$

$$P_{\downarrow} \underset{AC(+)}{=} Q_{\downarrow}$$

Let P be a well-formed process. Let $P_{\downarrow}$ be its normal form (whose existence and uniqueness comes from the termination (2) and confluence (3) of the rewriting system). With (4) and (5) we deduce that $P_{\downarrow}$ is in canonical form and by (1) that it is equivalent to P. $P_{\downarrow}$ is thus a perfect candidate for our theorem.

Now assume that there exists another process Q in canonical form that is equivalent to P. By lemma (6) we know that the normal form $Q_{\downarrow}$ of Q can be reached from Q by the reductions $A(;)$, $I(+)$ and $S(\mathsf{skip})$. Because of the confluence, $P_{\downarrow}$ and $Q_{\downarrow}$ which are in normal form are necessarily equal modulo $AC(+)$ (and $AC(||)$, but they do not contain any occurrence of $||$). Reusing (1) we find that $P_{\downarrow}$ and Q are equal modulo the expected axioms.

Details of the proof are given in appendix B. The termination of the rewriting system has been proved automatically using the rewriting tool *cime* [3]. We used the same tool to automatically generate and join most of the critical pairs that are the key to the local confluence proof. The remaining critical pairs were joined by hand.

Theorem 10.2 (Preservation of Consistency) *The canonical form of a locally consistent process is also a locally consistent process.*

Proof It is sufficient to show subject-reduction for locally consistent processes.

This is clear by inspection of the rules, because in no rule do we break any sequence of primitive actions in brackets. So, if a primitive action is part of a bracketed consistent sequence of primitive actions in the initial process, it will also be part of such a sequence in the final process. The only thing that can happen is that such sequences disappear, as with rules $\langle x\rangle_A \ ;\ \downarrow\uparrow \ \to \ \downarrow\uparrow \ ;\ \langle x\rangle_A \ +\ \downarrow\uparrow$ and $\langle \downarrow\uparrow \ ;\ x\rangle_A \ \to\ \mathsf{skip}$.

Conclusion

This paper has presented 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.

We hope that the work done in this paper can 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 proving that the machine satisfies all the equational axioms. This is left for future work.

Another continuation of this work would be to 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 processes presented in this paper are 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] Andrew P. Black. Understanding transactions in the operating system context. *Record of Fourth ACM SIGOPS European Workshop, Operating Systems Review*, 25(1):73–76, 1991. Workshop held in Bologna, Italy, September 1990.

[2] Philip Bernstein, Vassos Hadzilacos, and Nathan Goodman. Concurrency Control and Recovery in Database Systems. Addison-Wesley. 1987.

[3] Evelyne Contejean, Claude Marché, Benjamin Monate, and Xavier Urbain. Cime version 2, 2000. Prerelease available at `http://cime.lri.fr/`.

[4] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[5] http://java.sun.com/products/ejb/

[6] http://www-3.ibm.com/software/info1/websphere/index.jsp

[7] http://www.bea.com/framework.jsp?CNT=homepage_main.jsp&FP=/content

[8] http://www.microsoft.com/com/tech/MTS.asp

[9] http://www.arjuna.com/products/arjunats/features.html

[10] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *COMPSUR*, 15(4):287–317, 1983.

[11] David. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, volume 12 of *SIGPLAN Notices*, pages 128–137, 1977. Published as SIGPLAN (12) 3.

[12] G. Parrington and S. Schrivastava. Implementing concurrency control in reliable distributed object-oriented systems. In *Proceedings European Conference on Object-Oriented Programming (ECOOP'88)*, volume ? of *LNCS*, August 1988. Norway.

[13] 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.

[14] D. Detlefs, M. Herlihy and J. Wing. Inheritance of synchronization and recovery properties in Avalon/C++. IEEE Computer, December 1988.

[15] J. Eppinger, L. Mummert, and A. Spector. Camelot and Avalon: A Distributed Transaction Facility. Morgan Kaufmann Publishers, 1991.

[16] J. Gray and A. Reuter. Transaction Processing: Techniques and Concepts. Morgan Kaufman. 1992

[17] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support distributed programs. ACM Transactions on Programming Languages and Systems, July 1993.

[18] J.E.B Moss. Nested transactions: An approach to reliable distributed computing. MIT Press, March 1985.

[19] S. Shrivastava. Lessons Learned from building and using the Arjuna distributed programming system. Theory and Practice in Distributed Systems. K. Birman (ed). Springer Verlag (LNCS 938). 1994.

[20] A.Z Spector et al. Support for distributed transactions in the TABS prototype. IEEE Transactions on Software Engineering, 11 (6). June 1985.

[21] J. Wing. Decomposing and Recomposing Transactional Concepts. Object-Based Distributed Programming. R. Guerraoui, O. Nierstrasz and M.Riveill (eds). Springer Verlag (LNCS 791).

[22] A. Rubio. A Fully Syntactic AC-RPO. Proc. RTA-99, LNCS 1631, 133-147, 1999.

Appendices

A Rewriting system modulo equivalence

In order to prove that there is exactly one process in canonical form in each equivalence class (Theorem 10.1), it is easier to work with a rewriting system whose reflexive, symmetric, transitive closure coincides with the equational theory.

One simple way to do it is to turn every equational rule into a rewriting rule by orienting it. But the resulting system is not in general confluent, terminating and with normal forms that are canonical, the three properties that ensure our goal. In figure 1 we present a rewriting system that we can show to be equivalent to the equational theory and that satisfies the three properties mentioned above.

For reasons relating to the decidability of unification, rewriting tools can deal with operators that are both associative and commutative but not with operators that are associative only. So, we declared the parallel composition and alternation operators to be associative and commutative, but we converted the associativity axiom for sequential composition into an (oriented) rewrite rule.

The problem then is that there is no rule to reduce a process like a ; $(\downarrow\uparrow\ ; y)$: we would like to use the reduction pattern for a ; $\downarrow\uparrow$, but the orientation of the associativity rule forbids us from reducing $a\ ;\ (\downarrow\uparrow\ ;\ y)$ to $(a\ ;\ \downarrow\uparrow)\ ;\ y$. The solution that we adopted was to add, for each rule whose pattern is composed of a sequence, an extra rule to deal with these problematic cases.

Another problem is that the equational axiom x ; $\mathsf{skip} = x$ was turned into the rule x ; $\mathsf{skip} \rightarrow x$. This means that the trick presented in section 4, which permits us to interleave an isolated action $\langle P\rangle_I$ that has no continuation by considering its continuation to be skip, does not work any more. Once again the solution was to add two rules to take into account the problematic cases.

Reminder on rewriting systems modulo an equivalence

In our rewriting system we have a set of *axioms* E and a set of *rules* R. Axioms and rules are both just pairs of terms containing variables. These variables are treated as if they were universally quantified. We write $P \simeq Q$ for an axiom and $P \rightarrow Q$ for a rule.

A *substitution* is an idempotent function that maps variables to terms. We can also apply a substitution to a term in the usual way. We write $t\sigma$ if we want to express the application of the substitution σ to the term t.

A substitution is said to be *closed* if it maps variables to ground terms (*i.e.*, to terms without variables).

$t|_p$ will represent the sub-term of the term t at position p.

$t[s]_p$ will represent the term obtained by substituting in t the term s for the subterm $t|_p$.

Equality Equality $\simeq$ is the symmetric, reflexive and transitive closure of the relation that contains all pairs (s, t) for which there exists an axiom $e_1 \simeq e_2 \in E$, a substitution σ and a position p such that:

$$s|_p = e_1\sigma \qquad \text{and} \qquad t = s[e_2\sigma]_p$$

One-step reduction Similarly, we will write $s \rightarrow t$ if there exists a rule $l \rightarrow r \in R$, a substitution

σ and a position p such that:

$$s|_p = l\sigma \qquad \text{and} \qquad t = s[r\sigma]_p$$

General reduction We will write $s \rightsquigarrow t$ if and only if

$$s \simeq s' \to t' \simeq t$$

B Proofs

Lemma B.1

$$P = Q \qquad \textit{iff} \qquad P \leftrightsquigarrow^* Q.$$

Proof The direction $(\Rightarrow)$ is straightforward because all the axioms of the equational theory are also present in the rewriting system, either as axioms or as rules.

To prove the other direction $(\Leftarrow)$, it is sufficient to show that the rules added to the rewriting system, when considered as equivalences, are derivable in the equational theory.

Lemma B.2 *The rewriting system is terminating.*

Proof This is shown by using the associative-commutative recursive path ordering [22] (ACRPO) defined on the total order:

$$|| \; > \; \mathit{crash}() \; > \; ; \; > \; \langle\rangle_I \; > \; \langle\rangle_A \; > \; \langle\rangle_D \; > \; a \; > \; \mathsf{skip} \; > \; + \; > \downarrow\uparrow$$

We used the rewriting system tool *cime* [3] to automatically check that all rules are decreasing w.r.t. this order.

For the termination proof, we took $\downarrow\uparrow$ to be a symbol. Although in fact it is only an abbreviation, the termination of the system where $\downarrow\uparrow$ is a symbol implies trivially the termination of the system where it is a notation for $\mathit{crash}(\mathsf{skip})$.

Lemma B.3 *The rewriting system is confluent on well-formed closed terms.*

Proof As the rewriting system is terminating, it is sufficient to show that it is locally confluent. Once again we used *cime*, this time to compute the critical pairs and to try to join them by normalization. The tool found 903 critical pairs, among which only 29 were not joinable, among which only 17 were valid (issued from a well-formed process), among which only 8 were different. These 8 critical pairs are listed in the figure 2.

In fact the rewriting system is not confluent on arbitrary terms (ill-formed or containing variables). The critical pairs are not joinable but we can show by using some lemmas that if we replace the variables by arbitrary well-formed processes, then they can be joined.

Here we just list the arguments that we used to deal with the remaining critical pairs.

1. For every process P and interleavable process P', there exist two terms Q and R such that:

   (a) $\langle P\rangle_I \; ; P' \; \rightsquigarrow^* \; Q$, and

$$
\begin{aligned}
(x \mid\mid y) \mid\mid z &\simeq x \mid\mid (y \mid\mid z) \\
x \mid\mid y &\simeq y \mid\mid x \\
(x + y) + z &\simeq x + (y + z) \\
x + y &\simeq y + x
\end{aligned}
$$

$$
(x \,;\, y) \,;\, z \to x \,;\, (y \,;\, z)
$$

$$
\begin{aligned}
x + x &\to x \\
(x + y) \,;\, z &\to x \,;\, z + y \,;\, z \\
x \,;\, (y + z) &\to x \,;\, y + x \,;\, z \\
x \,;\, ((y + z) \,;\, t) &\to x \,;\, (y \,;\, t) + x \,;\, (z \,;\, t) \\
(x + y) \mid\mid z &\to x \mid\mid z + y \mid\mid z \\
\langle x + y \rangle_k &\to \langle x \rangle_k + \langle y \rangle_k \qquad k \in \{A, D, I\} \\
\mathit{crash}(x + y) &\to \mathit{crash}(x) + \mathit{crash}(y)
\end{aligned}
$$

$$
\begin{aligned}
\langle \mathsf{skip} \rangle_k &\to \mathsf{skip} \qquad k \in \{A, D, I\} \\
x \,;\, \mathsf{skip} &\to x \\
\mathsf{skip} \,;\, x &\to x \\
x \mid\mid \mathsf{skip} &\to x \\
\mathit{crash}(\mathsf{skip}) &\to \downarrow\uparrow
\end{aligned}
$$

$$
\begin{aligned}
\mathit{crash}(a) &\to \downarrow\uparrow \,;\, a + \downarrow\uparrow \\
\mathit{crash}(\langle x \rangle_A) &\to \downarrow\uparrow \,;\, \langle x \rangle_A + \downarrow\uparrow \\
\mathit{crash}(\langle x \rangle_D) &\to \downarrow\uparrow \,;\, \langle \mathit{crash}(x) \rangle_D \\
\mathit{crash}(\langle x \rangle_I) &\to \mathit{crash}(x) \\
\mathit{crash}(x \,;\, y) &\to \mathit{crash}(x) \,;\, \mathit{crash}(y) \\
\mathit{crash}(\mathit{crash}(x)) &\to \mathit{crash}(x) \\
\mathit{crash}(\downarrow\uparrow) &\to \downarrow\uparrow
\end{aligned}
$$

$$
\begin{aligned}
a \,;\, \downarrow\uparrow &\to \downarrow\uparrow \,;\, a + \downarrow\uparrow \\
a \,;\, (\downarrow\uparrow \,;\, y) &\to \downarrow\uparrow \,;\, (a \,;\, y) + \downarrow\uparrow \,;\, y \\
\langle x \rangle_A \,;\, \downarrow\uparrow &\to \downarrow\uparrow \,;\, \langle x \rangle_A + \downarrow\uparrow \\
\langle x \rangle_A \,;\, (\downarrow\uparrow \,;\, y) &\to \downarrow\uparrow \,;\, (\langle x \rangle_A \,;\, y) + \downarrow\uparrow \,;\, y \\
\langle x \rangle_D \,;\, \downarrow\uparrow &\to \downarrow\uparrow \,;\, \langle x \rangle_D \\
\langle x \rangle_D \,;\, (\downarrow\uparrow \,;\, y) &\to \downarrow\uparrow \,;\, (\langle x \rangle_D \,;\, y) \\
\langle x \rangle_I \,;\, \downarrow\uparrow &\to x \,;\, \downarrow\uparrow \\
\langle x \rangle_I \,;\, (\downarrow\uparrow \,;\, y) &\to x \,;\, (\downarrow\uparrow \,;\, y) \\
\downarrow\uparrow \,;\, \downarrow\uparrow &\to \downarrow\uparrow \\
\downarrow\uparrow \,;\, (\downarrow\uparrow \,;\, y) &\to \downarrow\uparrow \,;\, y
\end{aligned}
$$

$$
\begin{aligned}
\langle \downarrow\uparrow \,;\, x \rangle_A &\to \mathsf{skip} \\
\langle \downarrow\uparrow \rangle_A &\to \mathsf{skip} \\
\langle \downarrow\uparrow \,;\, x \rangle_D &\to \langle x \rangle_D \\
\langle \downarrow\uparrow \rangle_D &\to \mathsf{skip} \\
\langle \downarrow\uparrow \,;\, x \rangle_I &\to \langle x \rangle_I \\
\langle \downarrow\uparrow \rangle_I &\to \mathsf{skip}
\end{aligned}
$$

$$
\begin{aligned}
\langle x \rangle_I \,;\, x' \mid\mid \langle y \rangle_I \,;\, y' &\to \langle x \rangle_I \,;\, (x' \mid\mid \langle y \rangle_I \,;\, y') + \langle y \rangle_I \,;\, (y' \mid\mid \langle x \rangle_I \,;\, x') \\
\langle x \rangle_I \mid\mid (\langle y \rangle_I \,;\, y') &\to \langle x \rangle_I \,;\, (\langle y \rangle_I \,;\, y') + \langle y \rangle_I \,;\, (y' \mid\mid \langle x \rangle_I) \\
\langle x \rangle_I \mid\mid \langle y \rangle_I &\to \langle x \rangle_I \,;\, \langle y \rangle_I + \langle y \rangle_I \,;\, \langle x \rangle_I
\end{aligned}
$$

Figure 1: Rewriting system modulo equivalence

$\langle x\rangle_I \mid\mid y + \langle x\rangle_I ; y$	$\langle x\rangle_I \mid\mid y$
$\langle x\rangle_I \mid\mid y \mid\mid z + \langle x\rangle_I ; y \mid\mid z$	$\langle x\rangle_I \mid\mid y \mid\mid z$
$\langle x\rangle_I ; y \mid\mid z + \langle x\rangle_I ; (y \mid\mid z)$	$\langle x\rangle_I ; y \mid\mid z$
$\langle x\rangle_I ; y \mid\mid z \mid\mid t + \langle x\rangle_I ; (y \mid\mid z) \mid\mid t$	$\langle x\rangle_I ; y \mid\mid z \mid\mid t$
$\downarrow\uparrow ; \mathit{crash}(x)$	$\mathit{crash}(x)$
$\mathit{crash}(x) ; \downarrow\uparrow$	$\mathit{crash}(x)$
$x ; \downarrow\uparrow$	$\downarrow\uparrow ; (x ; \downarrow\uparrow)$
$x ; (\downarrow\uparrow ; y)$	$\downarrow\uparrow ; (x ; (\downarrow\uparrow ; y))$

Figure 2: Remaining critical pairs

(b) $\langle P\rangle_I \mid\mid P' \rightsquigarrow^* Q + R$ (or simply Q).

2. For every process P and interleavable processes P' and P'', there exist two terms Q and R such that:
   (a) $\langle P\rangle_I ; P' \mid\mid P'' \rightsquigarrow^* Q$, and
   (b) $(\langle P\rangle_I ; P') \mid\mid P'' \rightsquigarrow^* Q + R$ (or simply Q).
3. For every well-formed process P, there exists a process Q such that:
   (a) $\mathit{crash}(P) \rightsquigarrow^* Q$, and
   (b) $\downarrow\uparrow ; \mathit{crash}(P) \rightsquigarrow^* Q$.
4. For every process P, there exists a process Q such that:
   (a) $\mathit{crash}(P) \rightsquigarrow^* Q$, and
   (b) $\mathit{crash}(P) ; \downarrow\uparrow \rightsquigarrow^* Q$.

The proofs of the two first arguments are based on an interleaving lemma which says that every interleavable process reduces to a process which is an alternation of sequences of isolated actions whose inner processes are strictly smaller than the initial process.

The proof of the third argument is based on the fact that every process $\mathit{crash}(P)$ can be reduced to an alternation of processes starting with $\downarrow\uparrow$.

Finally the proof of the fourth argument is by induction on the process P and requires the interleaving lemma.

Lemma B.4 *Reduction preserves well-formedness.*

Proof We can show that a well-formed process can be reduced only to another well-formed process. This is a kind of subject-reduction property, but with a very simple notion of typing where there are only two available types: well-formed processes and interleavable processes.

It is sufficient to show that for each rule of the rewriting system, if the left-hand side is well-formed (resp. interleavable) then the right-hand side is also well-formed (resp. interleavable). For axioms we must check this property in both directions.

To give an idea of the content of the proof, let us examine a typical case: the rule $\langle x\rangle_I ; \downarrow\uparrow \rightarrow x ; \downarrow\uparrow$. Here the lhs cannot be interleavable because there is a $\downarrow\uparrow$ occurrence at the outermost level. So we have to consider just the case where the lhs is well-formed and show that the rhs is well-formed as well. If the lhs is well-formed, then by inverting the well-formedness predicate we see that x is also well-formed, which implies the well-formedness of $x ; \downarrow\uparrow$.

Lemma B.5 *Well-formed normal processes are in canonical form.*

Proof We show by mutual induction on the predicates *well-formed* and *interleavable* that a normal well-formed process is in canonical form and that a normal interleavable process is an alternation of sequences of isolated actions containing normal well-formed processes without $\downarrow\uparrow$ or skip. To see the kind of reasoning involved in this proof, let us again examine an example.

Assume that $P = I \mid\mid J$ is a normal well-formed process. P is normal, so necessarily I and J are normal as well. P is well-formed, so necessarily I and J are interleavable. By the induction hypothesis I and J are sequences of isolated actions. But then they can be interleaved using one of the rewriting rules that represent interleaving, contradicting the fact that P is normal.

Lemma B.6 *A process in canonical form can be reduced to a process in normal form using only the rules $A(;)$, $I(+)$ and $S(\mathsf{skip})$.*

Proof We observe that the A, I and S rules are the only ones that can be applied to a process in canonical form. This is because all + operators are at the top level, so the rules of distributivity of + are not applicable. Similarly, because there are no $\mid\mid$ operations, the interleaving rules are not applicable, and so on.