

Axiomatization of Transactions

Vincent Cremet, Andrew Black, Martin Odersky, Rachid Guerraoui

December 19, 2002

Abstract

A transaction is defined informally as a computation unit executing in an environment where there can be other transactions running in parallel and competing for the same shared resources, and which can be subjected to crashes that it can recover from. It is expected that the execution of a program composed of transactions will preserve the consistency of the system. One common way to meet this global property is to require from each transaction that it satisfies the four ACID properties: atomicity, (local) consistency, isolation and durability. The idea of this work is to give a more precise understanding of the concepts of transaction and failure by giving a small calculus in which ACID properties are decomposed. Armed with such a formalism we are then able to state and prove a theorem of consistency preservation.

Introduction

The central idea of this work is to find for each concurrent program that could fail during its execution a set of programs that can not fail, that are not concurrent and which includes the possible results of the first program. It is essentially a proof technique.

The problem is that we have no operational semantics for specifying the possible results of a given transaction.

Suppose we want to prove that the possible results of the execution of a program P satisfy all a given property. It is quite difficult to do in general because we have to take into account the possibilities of crash as well as some complex mechanisms of concurrency resolution.

Instead if we know that the possible results of the initial process P are included in the possible behaviours of a process Q that can not fail and which is not concurrent, it is simpler to prove the property on this last process.

The hope is that even if Q represents a larger set of results on which to prove the property, it will still be easier because of the simplification due to not taking into account crashes and concurrency. Also the fact of erasing of a larger set of possible results leaves place to a less restrictive implementation choice.

But how can we get this simpler process Q from the initial process P ?

One idea is to introduce new symbols in the syntax of the calculus to speak of crash/recovery events. $\text{crash}(t)$ will mean “one or more crashes during t ” and $t ; \downarrow\uparrow ; u$ will mean “One crash between t and u . Of course it is strange to have in the syntax symbols that represent events occurring at runtime.

But one thing is sure: if everything is serialized, it is easy to consider a crash/recovery as a primitive action with a special effect on the memory.

In the sentence $\pi_1 ; \pi_2 ; \downarrow\uparrow ; \pi_3$ it is clear what the symbol $\downarrow\uparrow$ means, namely that a crash/recovery event happens after π_2 and before π_3 and it is then easy to compute the set of possible results, here π_3 .

But in the opposite case, if some parts are not serialized it is more difficult to find a meaning of the symbol $\downarrow\uparrow$. What is the meaning of the sentence $t \& u \& \downarrow\uparrow$? It is exactly that a crash will occur during the parallel execution of t and u , what we choose to write $\text{crash}(t \& u)$. (Why? Explain.)

An example of such a property will be the consistency preservation. Using the proof principle given above we show it in one of our section.

The remainder of this paper is organized as follows: in the first section we give an intuitive description of the concepts we want to be able to capture in our calculus.

To be able to speak of these concepts in a formal setting we give us a language that is used to express at the same level the syntax of programs and the indication of a crash event during their execution.

In the next section we justify the idea of using a calculus rather than an equivalence relation, and what will be the philosophy behind the choice of the rules. In the next section we search in natural language some relations between programs that should behave if not the same at least in a similar way. Finally we recapitulate all the rules in a table.

We also present some results about our formalism: subject reduction, confluence, termination.

1 Intuitive background

In this section we present informally the intuitive notions that we want to formalize through our abstract calculus. It starts with the concept of *machine* on which *primitive actions* can be performed on global shared resources. Such machines can execute in parallel a particular kind of programs that we call *transactions*. These transactions should satisfy partially or totally the four *ACID* properties and can be arbitrarily *nested*. Finally, during the execution of a program *crash/recovery* events can occur.

A machine

We assume that we have a non reliable machine (a computer) which:

- can be loaded with a program and execute it.
- can crash during the execution of a program and then resume from the execution point it were when it crashed.
- has a hierarchy of memories (that can be in practice simulated by a single memory with a stack). A crash of the machine will be local to a given memory. The effect of such a crash will be to empty this memory, as well as all sub-memories in the hierarchy.
- has a persistent storage for each memory in the hierarchy that can resist crashes, and which is used as a backup for the memories during the recovery mechanism.
- has a shared global memory whose state can be updated by primitive actions.

Primitive actions

Primitive actions represent accesses to shared global resources of the system. Typically it will be a method call on a global object.

We call \mathcal{A} the set of primitive actions and use π , π_0 , etc to range over them.

A crash/recovery

A single crash/recovery event can be thought intuitively as an erasure of the local memory of a nested transaction (the crash) followed by a resuming of program execution in a context where the memory has been reinitialized with the content of the backup (the recovery).

We consider a crash recovery to be atomic. It means that we do not dissociate the crash phase from the recovery phase. We exclude for instance that a new crash can happen during the recovery phase.

Finally, the *severity* of a crash corresponds to the depth at which it occurs: the deeper it is the less severe it is.

A transaction

A transaction is a computation unit that can invoke primitive actions for modifying the system state and that can satisfy an arbitrarily subset of the four following properties, usually called *the ACID properties*:

- **atomicity**: either the transaction executes completely or not at all.
- **consistency**: in absence of crash and of other transactions running in parallel, the transaction execution preserves the consistency of the system.
- **isolation**: a parallel execution of isolated transactions must have the same effect as some sequential execution of the same transactions.

- **durability**: the effect of a durable transaction that has completed cannot be undone, even if a crash occurs (this property is also called persistence).

Transactions can be executed sequentially or in parallel.

Notice that we do not require in the definition of a transaction that it satisfies all the ACID properties as it is usually the case. It is an originality of this work to lighten the definition and to clearly decompose each property in the syntax of the calculus.

Also, to simplify, we do not take into account the concepts of *commit* or *abort*: we will consider that a transaction that completes will automatically commit to the directly enclosing transaction, and an abort can be simulated in a first approximation by a crash/recovery event inside a atomic transaction.

Nested transactions

We allow transactions to be nested inside a program and we assume that this nesting of transactions is interpreting as a nesting of associated memories during the program execution on a machine.

We use the brackets \langle and \rangle in a program to delimit a nested transaction.

So an explicit crash/recovery event inside a transaction should be interpreted as a crash/recovery event which is local to the memory associated with the transaction.

For instance in the transaction

$$\langle \langle P \rangle_I \ \& \ \langle Q ; \downarrow \uparrow \rangle_I \rangle_A$$

the crash/recovery will only affect the nested transaction containing Q , not the transaction containing P nor the top-level transaction. But a crash/recovery occurring in some transaction can affect nested transactions of this transaction according to our remark about the hierarchy of memories.

In fact brackets are used to represent two different concepts: the nesting of transactions and the ACID property that this transaction should satisfy.

Consistent sequences of primitive actions

Among all the possible sequences of primitive actions, we want to distinguish a subset that will contain only *consistent sequences*.

These sequences of actions are meant to preserve the global consistency of the system.

For instance suppose that we have at our disposal two kinds of primitive actions: ($a.withdraw(m)$) to withdraw an amount of money m from an account a , and ($a.deposit(m)$) to perform the reverse operation.

In this context the sequences of the shape $(a_1.withdraw(m), a_2.deposit(m))$ that moves an amount of money m from an account a_1 to an account a_2 will be consistent but not $(a_1.withdraw(m))$ or $(a_2.deposit(m))$ alone.

The implicit consistency property that has to be preserved in this case says that the sum of the amounts on both accounts must be constant.

But in the same way as we were not interested in the particular semantics of primitive actions, we are not interested here in the logical properties that have to be preserved by the execution of a program. We only assume that we have a set of primitive action sequences that individually preserve them.

So we give us a set of consistent sequences $\mathcal{C} \in \mathcal{A}^*$.

2 Language primitives

We want to define a formal language to express both the kind of programs that we want to execute and the parts of a computation that is likely to crash during its execution.

In fact we will mix voluntarily in a same language operators belonging to the standard level and some belonging to the meta-level of the language.

Some operators belong to the standard level, like $(\&)$ to express concurrency or like $(;)$ to express a sequencing of actions.

Some other operators belong to the meta-level, namely $\text{crash}()$ and $\downarrow\uparrow$. Usually you do not express in your program that you want some crash/recovery event to occur during a part of your code. First it is a bit weird to expect eagerly that something wrong occurs, secondly it is not something syntactically attached to a given text zone of the program, it is something that happens during the execution of a program.

We will try in this section to relate the primitive operator symbols of the language to the intuitive background presented in the previous section.

t, u	$::=$	π	primitive action
		$\langle t \rangle_A$	atomic transaction
		$\langle t \rangle_D$	durable transaction
		$\langle t \rangle_I$	isolated transaction
		$t \& u$	parallel composition
		$t ; u$	sequential composition
		$t + u$	non-deterministic choice
		skip	null action
		$\text{crash}(t)$	one or more crash/recoveries during t
		$\downarrow\uparrow$	a single perfectly located crash/recovery

(We could have split $\langle \cdot \rangle_k$ into two operators: $\langle \cdot \rangle$ for nested transactions and $k(\cdot)$ for “property” transactions.)

The precedence order for operators is: $\&$ $>$; $>$ $+$.

Explanations π ranges over primitives actions. The expressions $\langle t \rangle_A$, $\langle t \rangle_D$ and $\langle t \rangle_I$ are used to define nested transactions that specifies respectively atomicity, durability and isolation properties. The expression $t \& u$ denotes that t and u can evolve in parallel.

A transaction $\langle t \rangle_D$ makes sense only in a context where there is risks of crashes. We can interpret it the following way: when transaction t completes, changes it made are made persistent by storing them on a hard disk.

The intuitive meaning of $\text{crash}(T)$ is: "During the execution of the transaction T there will happen some crash/recoveries. The question of knowing if we assume at least one crash/recovery or if we allow none is not crucial because the case of the absence of crash can be simulated by one crash just before T . And it is natural to think that a crash just at the beginning has no observational effect, because nothing has been done yet.

3 Choice of formalization

This work aims at defining a concurrent programming language in which we can explicitly express in the syntax that a particular part of a program has to satisfy a subset of the ACID properties, as well as to give a precise semantics of the language.

We choose to give the semantics as a compositional reduction relation \prec modulo a congruence relation \simeq . But we do not interpret the reduction relation as it is usually done: in the reduction step $T \prec U$ we do not think at U as an expression simpler than T with a smaller set of possible results (as the step $\tau.P + Q \rightarrow P$ in the π -calculus). On the contrary, we think at U as a simpler expression with a bigger set of possible results, hence the symbol \prec chosen for the relation.

The idea behind this counter-intuitive meaning is essentially a proof technique: if we want to prove that a property is satisfied by all the possible results of T and we know that $T \prec U$, we can just prove that the property is satisfied by all the possible results of the *simpler* program U . In our case simpler will mean less concurrent and/or with a more precise knowledge about the location where the crash/recovery events occur. The implicit hope is that if the program is simpler the proof should be easier.

We will use this proof principle later to prove the consistency preservation property of consistent programs.

T denotes the possible results of executing T .

4 Discovering of the rules

We plan in this section to determine a set of reduction and equivalence rules based on common sense and to give for most of them an intuitive

explanation.

Here is an overview of the rules. We divide the rules into several logical categories.

Simple associativity and commutativity

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

It is not surprising that parallel composition is associative and commutative. It is not always the case for $+$ (Cf. bisimulation in the π -calculus where $+$ is not associative), but here it is because we do not discard alternatives. The sequential composition of processes is of course associative.

Distribution of the choice operator

$$\begin{aligned}
x + x &\prec x \\
(x + y) ; z &\prec x ; z + y ; z \\
x ; (y + z) &\prec x ; y + x ; z \\
(x + y) \& z &\prec x \& z + y \& z \\
\langle x + y \rangle_k &\prec \langle x \rangle_k + \langle y \rangle_k \quad k \in \{A, D, I\} \\
\text{crash}(x + y) &\prec \text{crash}(x) + \text{crash}(y)
\end{aligned}$$

The choice operator distributes over every other operator. That only says that a choice at a local level determines a choice for the entire program.

Simplification involving the non-informative symbol *skip*

$$\begin{aligned}
\langle \text{skip} \rangle_k &\prec \text{skip} \quad k \in \{A, D, I\} \\
x ; \text{skip} &\prec x \\
\text{skip} ; x &\prec x \\
x \& \text{skip} &\prec x \\
\text{crash}(\text{skip}) &\prec \downarrow\uparrow
\end{aligned}$$

The symbol *skip* represents an action that has no effect, so there is no reason why a programmer would use it in his program. But it is produced only at one place in our rules, namely as the result of an “aborted” atomic transaction. It is used only because we want to treat processes compositionally. So a *skip* will temporarily fill a hole left by an aborted atomic transaction, before being removed by the enclosing transaction through the logical rules presented above.

Precising the location of a crash

In the rules of the form $\text{crash}(t) \prec u$ the term u is simpler in the sense that we know more accurately where the crash/recoveries foreseen by $\text{crash}()$ will take place.

$$\begin{array}{lcl}
 \text{crash}(A) & \prec & \downarrow\uparrow ; A + \downarrow\uparrow \\
 \text{crash}(\langle x \rangle_A) & \prec & \downarrow\uparrow ; \langle x \rangle_A + \downarrow\uparrow \\
 \text{crash}(\langle x \rangle_D) & \prec & \downarrow\uparrow ; \langle \text{crash}(x) \rangle_D \\
 \text{crash}(\langle x \rangle_I) & \prec & \text{crash}(x) \\
 \text{crash}(x ; y) & \prec & \text{crash}(x) ; \text{crash}(y) \\
 \text{crash}(\text{crash}(x)) & \prec & \text{crash}(x) \\
 \text{crash}(\downarrow\uparrow) & \prec & \downarrow\uparrow
 \end{array}$$

$\text{crash}(T ; U) \prec \text{crash}(T) ; \text{crash}(U)$: "If some crash/recoveries occurred during the sequence of T followed by U, then they can have occurred during T or/and during U." It is correct with the intuitive interpretation of \prec and $\text{crash}()$.

It is required to have this rule because a symbol $;$ can in fact be the result of an interleaving involving two processes that can have crashed simultaneously.

We could have written:

$$\text{crash}(T ; U) \prec \text{crash}(T) ; \text{crash}(U) + \text{crash}(T) ; U + T ; \text{crash}(U)$$

But we can prove that $\text{crash}(T) ; U$ and $T ; \text{crash}(U)$ are subsumed by $\text{crash}(T) ; \text{crash}(U)$ because:

- 1) $\text{crash}(U)$ contains the case $\downarrow\uparrow ; U$ (so $\text{crash}(T) ; \text{crash}(U)$ contains $\text{crash}(T) ; \downarrow\uparrow ; U$), and we can show that $\text{crash}(T) ; \downarrow\uparrow$ is "equivalent" to $\text{crash}(T)$.
- 2) $\text{crash}(T)$ contains the case $T ; \downarrow\uparrow$ and we can show that $\downarrow\uparrow ; \text{crash}(U)$ is equivalent to $\text{crash}(U)$.

$\text{crash}(\langle T \rangle_k)$: a crash/recovery local to the transaction enclosing $\langle T \rangle_k$. It can also affect T .

$$\text{crash}(\langle x \rangle_D) \prec \downarrow\uparrow ; \langle \text{crash}(x) \rangle_D : ?$$

$\text{crash}(\langle T \rangle_I) \prec \text{crash}(T)$: Same as for $\langle t \rangle_I$; $\downarrow\uparrow \prec t$; $\downarrow\uparrow$ (See below).

Effect of a crash/recovery on already resolved actions

$$\begin{array}{l}
\pi ; \downarrow\uparrow \prec \downarrow\uparrow ; \pi + \downarrow\uparrow \\
\langle x \rangle_A ; \downarrow\uparrow \prec \downarrow\uparrow ; \langle x \rangle_A + \downarrow\uparrow \\
\langle x \rangle_D ; \downarrow\uparrow \prec \downarrow\uparrow ; \langle x \rangle_D \\
\langle x \rangle_I ; \downarrow\uparrow \prec x ; \downarrow\uparrow \\
\downarrow\uparrow ; \downarrow\uparrow \prec \downarrow\uparrow
\end{array}$$

$\langle x \rangle_D ; \downarrow\uparrow \prec \downarrow\uparrow ; \langle x \rangle_D$: A crash can affect in principle every action that precedes it. As the transaction is durable, the crash/recovery that follows it does not affect it, but it could also affect things that happened before this transaction. So we report it before.

$\downarrow\uparrow ; \downarrow\uparrow \prec \downarrow\uparrow$: "One crash/recovery or several crash/recoveries have exactly the same effect".

It comes from the intuitive meaning of a crash/recovery. It is clear that a sequence of any number of such an operation will give the same result as a sequence of length one. Because we can see a crash/recovery as a procedure that replaces the memory state by the contents of the backup:

$$\langle T \rangle_I ; \downarrow\uparrow \prec T ; \downarrow\uparrow \text{ and } \text{crash}(\langle T \rangle_I) \prec \text{crash}(T).$$

What allows us to skip the isolation brackets here is the knowledge that $\langle T \rangle_I$ is no more interleavable, since it is at the same level as a $\downarrow\uparrow$ in the first case and as a $\text{crash}()$ in the second case. So the processes in the left members of the rules are not well-formed interleavable processes, so the isolation brackets are no longer useful.

$$\langle T \rangle_A ; \downarrow\uparrow \prec \downarrow\uparrow ; \langle T \rangle_A + \downarrow\uparrow$$

Effect of a crash/recovery at the beginning of a nested transaction

$$\begin{array}{l}
\langle \downarrow\uparrow ; x \rangle_A \prec \text{skip} \\
\langle \downarrow\uparrow ; x \rangle_D \prec \langle x \rangle_D \\
\langle \downarrow\uparrow ; x \rangle_I \prec \langle x \rangle_I
\end{array}$$

$\langle \downarrow\uparrow ; T \rangle_D \prec \langle T \rangle_D$: "If a local crash/recovery occurs at the beginning of a transaction it has no effect, because the local memory and the local backup are still empty (Cf. Intuitive meaning of a crash/recovery).

A crash/recovery is local to a nested transaction, so it does not extend beyond the nearest enclosing brackets.

Remark: Our model allows a sub-transaction of an atomic transaction to fail without making the whole transaction to abort.

For instance we have

$$\langle t ; \langle \downarrow \uparrow ; u \rangle_A \rangle_A \prec \langle t \rangle_A$$

Concurrency resolution

$$\begin{aligned} \langle \langle x \rangle_I ; x' \rangle &\& \langle \langle y \rangle_I ; y' \rangle &\prec &\langle \langle x \rangle_I ; x' \rangle &\& \langle \langle y \rangle_I ; y' \rangle \\ &+ && &\langle \langle y \rangle_I ; y' \rangle &\& \langle \langle x \rangle_I ; x' \rangle \\ \langle \langle x \rangle_I \rangle &\& \langle \langle y \rangle_I ; y' \rangle &\prec &\langle \langle x \rangle_I ; \langle \langle y \rangle_I ; y' \rangle \rangle \\ &+ && &\langle \langle y \rangle_I ; y' \rangle &\& \langle \langle x \rangle_I \rangle \\ \langle \langle x \rangle_I \rangle &\& \langle \langle y \rangle_I \rangle &\prec &\langle \langle x \rangle_I ; \langle \langle y \rangle_I \rangle \rangle \\ &+ && &\langle \langle y \rangle_I ; \langle \langle x \rangle_I \rangle \rangle \end{aligned}$$

$\langle \langle x \rangle_I \rangle &\& \langle \langle y \rangle_I \rangle \prec \langle \langle x \rangle_I ; \langle \langle y \rangle_I \rangle + \langle \langle y \rangle_I ; \langle \langle x \rangle_I \rangle$: “A parallel composition of two isolated transactions must be equivalent to some sequential composition of these transactions”.

It is precisely what we usually mean by *isolation*.

In the case where things to interleave are arbitrary long sequences of isolated transactions, and not simply single isolated transactions, we can use one of the other rules:

$$\begin{aligned} \langle \langle \langle x \rangle_I ; x' \rangle \rangle &\& \langle \langle \langle y \rangle_I ; y' \rangle \rangle &\prec &\langle \langle \langle x \rangle_I ; x' \rangle \rangle &\& \langle \langle \langle y \rangle_I ; y' \rangle \rangle \\ &+ && &\langle \langle \langle y \rangle_I ; y' \rangle \rangle &\& \langle \langle \langle x \rangle_I ; x' \rangle \rangle \\ \langle \langle \langle x \rangle_I \rangle \rangle &\& \langle \langle \langle y \rangle_I ; y' \rangle \rangle &\prec &\langle \langle \langle x \rangle_I ; \langle \langle \langle y \rangle_I ; y' \rangle \rangle \rangle \\ &+ && &\langle \langle \langle y \rangle_I ; y' \rangle \rangle &\& \langle \langle \langle x \rangle_I \rangle \rangle \end{aligned}$$

Some lacking rules

Remark there is no rule for reducing $\text{crash}(\langle \rangle X \& Y)$ nor $X \& Y ; \downarrow \uparrow$, nor $\text{crash}(X) ; \downarrow \uparrow$.

For $\text{crash}(\langle \rangle X \& Y)$, we prefer reduce first $X \& Y$ into a sequence and then apply the crash/recovery to each component of the sequence. Naively we could have reduce it to $\text{crash}(X) \& \text{crash}(Y)$ but then we should have been able to interleave crash/recoveries, something that has revealed difficult, if not impossible (see section Historic).

We adopt the same idea for $X \& Y ; \downarrow \uparrow$.

For $\text{crash}(X) ; \downarrow \uparrow$, we could have reduced it to $\text{crash}(X)$ because we can show that these two processes can reduce to a same process for any subprocess X .

5 Some results

We have shown some meta-theoretical properties of our calculus, namely subject reduction, termination and confluence. All together these results allow us to speak of the unique normal form of a given well-formed program. Armed with this fact and our interpretation of the reduction relation of the calculus we are able to show one of our ultimate goal which is the preservation of the global consistency of a system by the execution of a well-formed program.

The rewriting system preserves well-formedness

We have a very simple notion of well-formedness of in our calculus when require a process to be interleavable if it has to be executed in parallel with others. Roughly speaking an interleavable process is a term of the calculus that is built only from transactions enclosed in isolation brackets. This kind of brackets allows to 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 .

An also important thing about the well-formedness predicate is that it excludes every occurrence of a crash/recovery at the point where an interleaving could take place. It comes from the difficulty to speak of the interleaving of a crash/recovery (see section historic for more details).

We define formally the set of processes and interleavable processes by mutual induction.

$$\begin{array}{l}
 P, Q, R \quad ::= \quad \textit{Process} \\
 \quad \quad \quad | \quad A \\
 \quad \quad \quad | \quad \langle P \rangle_k \quad k \in \{A, D, I\} \\
 \quad \quad \quad | \quad I \ \& \ J \\
 \quad \quad \quad | \quad P ; Q \\
 \quad \quad \quad | \quad P + Q \\
 \quad \quad \quad | \quad \textit{crash}(P) \\
 \quad \quad \quad | \quad \textit{skip} \\
 \\
 I, J \quad ::= \quad \textit{Interleavable processes} \\
 \quad \quad \quad | \quad \langle P \rangle_I \\
 \quad \quad \quad | \quad I \ \& \ J \\
 \quad \quad \quad | \quad I ; J \\
 \quad \quad \quad | \quad I + J \\
 \quad \quad \quad | \quad \textit{skip}
 \end{array}$$

Notice that they are ground terms (without variables), and that interleavable processes are processes.

We can show that a well-formed process can only reduce to a well-formed process, it is a kind of subject reduction statement but with a very simple notion of typing where there are only to available types: well-formed processes and well-formed interleavable processes.

We have also the important property that a well-formed *consistent* process reduces to a well-formed *consistent* process. But it would not hold any more if we allowed crash/recoveries inside consistent sequences. The consistency preservation theorem would be still valid.

The rewriting system is terminating

PROOF. It is shown very simply using the associative-commutative recursive path ordering (ACRPO) defined on the total order:

1.
 - $\langle x \rangle_I \& y + \langle x \rangle_I ; y$
 - $\langle x \rangle_I \& y$
2.
 - $\langle x \rangle_I \& y \& z + (\langle x \rangle_I ; y) \& z$
 - $\langle x \rangle_I \& y \& z$
3.
 - $(\langle x \rangle_I ; y) \& z + \langle x \rangle_I ; y \& z$
 - $(\langle x \rangle_I ; y) \& z$
4.
 - $(\langle x \rangle_I ; y) \& z \& t + (\langle x \rangle_I ; y \& z) \& t$
 - $(\langle x \rangle_I ; y) \& z \& t$
5.
 - $\downarrow\uparrow ; \text{crash}(x)$
 - $\text{crash}(x)$
6.
 - $\text{crash}(x) ; \downarrow\uparrow$
 - $\text{crash}(x)$
7.
 - $x ; \downarrow\uparrow$
 - $\downarrow\uparrow ; (x ; \downarrow\uparrow)$
8.
 - $x ; (\downarrow\uparrow ; y)$
 - $\downarrow\uparrow ; (x ; (\downarrow\uparrow ; y))$

Figure 1: Remaining critical pairs

$\& > \text{crash}() > ; > \langle \rangle_I > \langle \rangle_A > \langle \rangle_D > A > \text{skip} > + > \downarrow\uparrow$

We used the rewriting system tool "cime" to automatically check the decreasing of all the rules w.r.t. this order.

The rewriting system is confluent on well-formed closed terms

PROOF. As the rewriting system is terminating, it is sufficient to show it is locally confluent. Once again we used "cime" to compute the critical pairs and try to join them by normalizing them. The tool found 903 critical pairs, among which only 29 were not joinable, among which only 17 were valid (issued from a well-formed transaction), among which only 8 were different:

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

The termination and confluence results allow us to define the function which maps each transaction to its unique normal form. Further-

more we have a practical means of computing a result for any input: any rewriting tool can do it.

Shape of normal forms

We can show easily by simple induction over process structure that an irreducible well-formed process contains neither parallel composition nor embedded failures. We can thus reason on such simpler processes and then extend the results about all well-formed processes if we accept the thesis exposed in section [?].

We are able to write a theorem of consistency preservation

Informally this theorem tells that every transaction built from consistent bricks is the refinement of another transaction that clearly preserve consistency.

This theorem tells also that for all process that can crash during its execution, there exists an equivalent process without parallel composition that behaves the same if it does not crash. So we can abstract from any crashes using this former process and prove more easily some interesting properties, like the preservation of consistency which is the subject of the next section.

Ultimate goal

How to show that a given implementation of transactions that decompose the properties A, D, I can execute a transaction while preserving the consistency of the system, even in presence of crash/recoveries and parallel execution of processes competing for shared resources ?

In fact this calculus can help us to prove things like that. It comes with an intuitive proof scheme which is the following.

If you can informally argue that your implementation satisfied the axioms and rules of our system, then it is more likely to preserve the consistency of the system. Unfortunately we did not succeed in formalize such an idea. We tried to define some kind of machine that performs computational steps and such that a crash/recovery can happen every time during this process, but it was quite difficult to relates it formally with our calculus.

So this axiomatization of transactions is more like a proof-technique used to demonstrate the consistency of an implementation of transactions.

6 Difficulty to go from theory to practice

7 Historic

Why a “global” crash/recovery symbol ($\text{crash}()$) and not simply $\downarrow\uparrow$.

Initial calculus

At the beginning there was no crash/recovery operator symbol $\text{crash}()$. Instead of using $\text{crash}(T)$ to express that a crash/recovery would occur during a transaction T we used the expression $\downarrow\uparrow \ \& \ T$ (which is no longer well-formed in the current version of the calculus).

It was surely attractive to have not to introduce an extra symbol but we met some problems as is explained below.

In fact the additional symbol was introduced due to the difficulty of speaking of a crash/recovery occurring during the parallel execution of several processes.

Even if it is quite natural to see a crash recovery as a kind of primitive actions, it quickly leads to some fundamental problems.

First remark:

Intuitively a crash/recovery can occur at any time during the execution of a program, it is not an *instruction* in a program. For instance:

7.1 Problem in the interaction between the interleaving rule and the distribution of a crash over a sequence

Let us consider the program:

$$\langle X \rangle_I \ \& \ \langle Y \rangle_I \ \& \ \downarrow\uparrow$$

It is difficult to include crash/recoveries at a high-level of reasoning because their effects depend completely on the implementation.

Example

$$\langle P \rangle \ \& \ \langle Q \rangle \ \& \ \downarrow\uparrow$$

With the previous axioms,

- we first interleave both transactions, e.g.:

$$(\langle P \rangle ; \langle Q \rangle) \ \& \ \downarrow\uparrow$$

- then we choose a victim, e.g.:

$$\langle P \rangle \ \& \ \downarrow\uparrow ; \langle Q \rangle$$

- We can see that the $\downarrow\uparrow$ happens either during P or during Q .

- But in a real implementation, we want two parallel transactions to overlap in order to increase concurrency. So a $\downarrow\uparrow$ could happen during both P and Q .
- **Possible solution:** reinterpret $P \ \& \ \downarrow\uparrow$ as: “0 or more crash/recoveries during P ” instead of “exactly one”.

$$(P ; Q) \ \& \ \downarrow\uparrow \equiv P \ \& \ \downarrow\uparrow ; Q + P ; Q \ \& \ \downarrow\uparrow$$

becomes

$$(P ; Q) \ \& \ \downarrow\uparrow \equiv P \ \& \ \downarrow\uparrow ; Q \ \& \ \downarrow\uparrow$$

Before we had to understand things like

Initial system

In one of the first systems, we had the two rules:

$$\begin{aligned} (T ; Q) \ \& \ \downarrow\uparrow &< T \ \& \ \downarrow\uparrow ; U + T ; U \ \& \ \downarrow\uparrow \\ \langle T \rangle_I \ \& \ \langle U \rangle_I &< \langle T \rangle_I ; \langle U \rangle_I + \langle U \rangle_I ; \langle T \rangle_I \end{aligned}$$

But we have the following problem when dealing with the process $\langle T \rangle_I \ \& \ \langle U \rangle_I \ \& \ \downarrow\uparrow$.

If we do the interleaving between isolated transactions T and U first, we get $\langle T \rangle_I ; \langle U \rangle_I + \langle U \rangle_I ; \langle T \rangle_I$. Next we can distribute the crash/recovery getting:

$$\langle T \rangle_I \ \& \ \downarrow\uparrow ; \langle U \rangle_I + \langle T \rangle_I ; \langle U \rangle_I \ \& \ \downarrow\uparrow + \langle U \rangle_I \ \& \ \downarrow\uparrow ; \langle T \rangle_I + \langle U \rangle_I ; \langle T \rangle_I \ \& \ \downarrow\uparrow$$

The important thing to notice is that either $\downarrow\uparrow$ occurs during T or during U but not during both, which would be possible if we consider that an implementation of transactions can allow some parallel transactions to overlap.

So our formalization was not expressive enough.

8 Examples demonstrating the expressive power of our calculus

Crash during two processes in parallel

In this section we will show how our model can handle the case of two processes executing in parallel.

The question is to find a set of possible behaviors that contains the behaviors of two processes in parallel that can suffer crashes.

We argue that the two rules:

$$\begin{aligned} \text{crash}(x ; y) &< \text{crash}(x) ; \text{crash}(y) \\ \langle x \rangle_I \ \& \ \langle y \rangle_I &< \langle x \rangle_I ; \langle y \rangle_I \\ &+ \langle y \rangle_I ; \langle x \rangle_I \end{aligned}$$

are enough to catch all the possibilities.

From these rules, we can derive

$$\begin{aligned}
\text{crash}(\langle X \rangle_I \ \& \ \langle Y \rangle_I) &\prec \text{crash}(\langle X \rangle_I ; \langle Y \rangle_I + \langle Y \rangle_I ; \langle X \rangle_I) \\
&\prec \text{crash}(\langle X \rangle_I ; \langle Y \rangle_I) + \text{crash}(\langle Y \rangle_I ; \langle X \rangle_I) \\
&\prec \text{crash}(\langle X \rangle_I) ; \text{crash}(\langle Y \rangle_I) + \text{crash}(\langle Y \rangle_I) ; \text{crash}(\langle X \rangle_I)
\end{aligned}$$

In a real implementation of transactions, two parallel transactions can overlap in order to increase concurrency, even if they were meant to be isolated. It will happen for example if the sets of resources accessed by each of the transactions are disjoint.

Only we must have the impression at the end that one of the transactions completed its execution before the other one started to execute.

We have to take into account 2 main cases. Either the crash occurred during the overlapping of the transactions, either it occurs during the first one before the second one started to execute.

In the first case, both transactions must have been affected by the crash, but for an external observer one transaction (e.g. X) must have executed completely before the second started (e.g. Y). So we should observe something like:

$$\text{crash}(\langle X \rangle_I) ; \text{crash}(\langle Y \rangle_I) \quad (1)$$

In the other cases, the crash occurred during one of the transaction, (but it does not mean that only one is affected). So we should observe something like:

$$\text{crash}(\langle X \rangle_I) ; \langle Y \rangle_I \quad (2)$$

or

$$\langle X \rangle_I ; \text{crash}(\langle Y \rangle_I) \quad (3)$$

In our calculus we can show that (2) and (3) are special cases of (1).

To see why let us proceed by phases.

It is not counter intuitive to admit that $\downarrow\uparrow ; \langle Y \rangle_I$ a special case of $\text{crash}(\langle Y \rangle_I)$, because if a crash can occur during $\langle Y \rangle_I$ it may be that it occurs just before $\langle Y \rangle_I$.

For similar reasons $\langle X \rangle_I ; \downarrow\uparrow$ will be a special case of $\text{crash}(\langle X \rangle_I)$.

So $\text{crash}(\langle X \rangle_I) ; \downarrow\uparrow ; \langle Y \rangle_I$ is a special case of $\text{crash}(\langle X \rangle_I) ; \text{crash}(\langle Y \rangle_I)$.

And we can conclude because in the calculus $\text{crash}(x) ; \downarrow\uparrow$ and $\text{crash}(x)$ are equivalent for all x . Of course it can seem surprising but here is an intuitive justification:

In our rules we always include the particular case in which the crash/recovery seems not to affect at all the transaction. There is no harm at that because it is reasonable to design a program so that it will behave soundly if there is no crash. On the contrary it would be insane that the programmer write something inconsistent with the hope that something wrong will happen and re-establish the consistency.

By doing that we obtain the property above because we can always “move” in an harmless way the ending crash/recovery to the point where the crash expressed by the $\text{crash}(\cdot)$ operator would happen.

8.1 The X symbol to make transaction kinds commutative

Currently transaction kinds are not commutative.

For instance: $\langle\langle X \rangle_I \rangle_A = \langle\langle X \rangle_A \rangle_I$ does not hold, because $\langle\langle X \rangle_I \rangle_A$ can not be interleaved, whereas $\langle\langle X \rangle_A \rangle_I$ can.

There is the same remark for $\langle \rangle_I$ with $\langle \rangle_D$.

For (A and D), there is a counter-example:

$\langle\langle X \rangle_A \rangle_D ; \downarrow \uparrow \prec \downarrow \uparrow ; \langle\langle X \rangle_A \rangle_D$ but $\langle\langle X \rangle_D \rangle_A ; \downarrow \uparrow \prec \downarrow \uparrow ; \langle\langle X \rangle_D \rangle_A + \downarrow \uparrow$

So, what we could call a *complete* transaction should be of the form:

$$\langle\langle\langle X \rangle_A \rangle_D \rangle_I$$

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, (3) it executes completely or not at all, because it is tagged a .

So, in $\langle\langle\langle X \rangle_A \rangle_D \rangle_I$ the order of the brackets is important and we can not use an notation $\langle X \rangle_T$ for any order of the brackets. It is the charge of the programmer to put the brackets in the order that will make the program behaves as he expects. This lost of flexibility is compensated by a simpler way to write the rules. In order to gain associativity of the transaction kinds we should add a deletion symbol (let's say X) that remember if something has really been damaged inside a atomic transaction. If it is not the case there is no need to abort the whole transaction.

8.2 Why do not take a symmetric relation ?

Remember that this work can be seen as a proof technique to show properties on transactions. We can see that a non symmetric relation can be very helpful if it is used through the following scheme proof:

We want to prove that each result of the execution of a given transaction T satisfies some property P , even in presence of interaction between parallel execution of processes and crashes.

If we can show that for all T and S in the reduction relation ($T \prec S$) the set of results of T is included in the set of results of S , and that for all result r of S we have $P(r)$ then for all result r of T we will have $P(r)$ as well.

It is exactly what we used to prove the consistency theorem.

An advantage of the deletion X symbol

Without the X symbol we have:

$$\langle\langle\text{crash}(;)A \rangle_A ; B \rangle_A \equiv \langle\text{skip} ; B \rangle_A$$

i.e. we have an atomic transaction which gets damaged by a crash/recovery but which is not aborted ...

Is really what we want ? Yes it is.

9 Conclusion

We have been working here on separating the properties of the transaction concept (Atomicity, Consistency, Isolation, and Durability) and precisely capturing the semantics of each of these properties. We devised a calculus of transactions and proved that every program in the calculus is equivalent to exactly one normal form, which contains neither parallel composition nor embedded failures. We thus are able to validate the claim of transactions that atomicity and isolation imply consistency.

A Table of

Contents

1	Intuitive background	2
2	Language primitives	5
3	Choice of formalization	6
4	Discovering of the rules	6
5	Some results	10
6	Difficulty to go from theory to practice	14
7	Historic	14
7.1	Problem in the interaction between the interleaving rule and the distribution of a crash over a sequence	14
8	Examples demonstrating the expressive power of our calculus	15
8.1	The X symbol to make transaction kinds commutative .	17
8.2	Why do not take a symmetric relation ?	17
9	Conclusion	18
A	Table of	18
B	The rewriting system	18
C	Summary of the rules	19

B The rewriting system

Basic notions

In our rewriting system we have a set of *axioms* E and a set of *rules* R . Axioms and rules are just pairs of terms with variables. We write

$P = Q$ for an axiom and $P \prec Q$ for a rule.

A *substitution* is a 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 (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 $t|_p$.

Simple equality

We will write $s = t$ if there exists an axiom $e1 = e2 \in E$, a substitution σ and a position p such that:

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

or

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

.

Simple reduction

Similarly, we will write $s \prec t$ if there exists a rule $l \prec r \in R$, a substitution σ and a position p such that:

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

.

General reduction

We will write $s \prec t$ if:

$$s =^* s' \prec t' =^* t$$

.

C Summary of the rules

$$(x \& y) \& z \simeq x \& (y \& z)$$

$$x \& y \simeq y \& x$$

$$(x + y) + z \simeq x + (y + z)$$

$$x + y \simeq y + x$$

$$(x ; y) ; z \prec x ; (y ; z)$$

$$x + x \prec x$$

$$(x + y) ; z \prec x ; z + y ; z$$

$$x ; (y + z) \prec x ; y + x ; z$$

$$x ; ((y + z) ; t) \prec x ; (y ; t) + x ; (z ; t)$$

$$(x + y) \& z \prec x \& z + y \& z$$

$$\langle x + y \rangle_k \prec \langle x \rangle_k + \langle y \rangle_k \quad k \in \{A, D, I\}$$

$$\text{crash}(x + y) \prec \text{crash}(x) + \text{crash}(y)$$

$$\langle \text{skip} \rangle_k \prec \text{skip} \quad k \in \{a, d, i\}$$

$$x ; \text{skip} \prec x$$

$$\text{skip} ; x \prec x$$

$$x \& \text{skip} \prec x$$

$$\text{crash}(\text{skip}) \prec \downarrow\uparrow$$

$$\text{crash}(A) \prec \downarrow\uparrow ; A + \downarrow\uparrow$$

$$\text{crash}(\langle x \rangle_A) \prec \downarrow\uparrow ; \langle x \rangle_A + \downarrow\uparrow$$

$$\text{crash}(\langle x \rangle_D) \prec \downarrow\uparrow ; \langle \text{crash}(x) \rangle_D$$

$$\text{crash}(\langle x \rangle_I) \prec \text{crash}(x)$$

$$\text{crash}(x ; y) \prec \text{crash}(x) ; \text{crash}(y)$$

$$\text{crash}(\text{crash}(x)) \prec \text{crash}(x)$$

$$\text{crash}(\downarrow\uparrow) \prec \downarrow\uparrow$$

$$A ; \downarrow\uparrow \prec \downarrow\uparrow ; A + \downarrow\uparrow$$

$$A ; (\downarrow\uparrow ; y) \prec \downarrow\uparrow ; (A ; y) + \downarrow\uparrow ; y$$

$$\langle x \rangle_A ; \downarrow\uparrow \prec \downarrow\uparrow ; \langle x \rangle_A + \downarrow\uparrow$$

$$\langle x \rangle_A ; (\downarrow\uparrow ; y) \prec \downarrow\uparrow ; (\langle x \rangle_A ; y) + \downarrow\uparrow ; y$$

$$\langle x \rangle_D ; \downarrow\uparrow \prec \downarrow\uparrow ; \langle x \rangle_D$$

$$\langle x \rangle_D ; (\downarrow\uparrow ; y) \prec \downarrow\uparrow ; (\langle x \rangle_D ; y)$$

$$\langle x \rangle_I ; \downarrow\uparrow \prec x ; \downarrow\uparrow$$

$$\langle x \rangle_I ; (\downarrow\uparrow ; y) \prec x ; (\downarrow\uparrow ; y)$$

$$\downarrow\uparrow ; \downarrow\uparrow \prec \downarrow\uparrow$$

$$\downarrow\uparrow ; (\downarrow\uparrow ; y) \prec \downarrow\uparrow ; y$$

$$\langle \downarrow\uparrow ; x \rangle_A \prec \text{skip}$$

$$\langle \downarrow\uparrow \rangle_A \prec \text{skip}$$

$$\langle \downarrow\uparrow ; x \rangle_D \prec \langle x \rangle_D$$

$$\langle \downarrow\uparrow \rangle_D \prec \text{skip}$$

$$\langle \downarrow\uparrow ; x \rangle_I \prec \langle x \rangle_I$$

$$\langle \downarrow\uparrow \rangle_I \prec \text{skip}$$

$$(\langle x \rangle_I ; x') \& (\langle y \rangle_I ; y') \prec \langle x \rangle_I ; x' \& (\langle y \rangle_I ; y')$$

$$+ \langle y \rangle_I ; y' \& (\langle x \rangle_I ; x')$$

$$\langle x \rangle_I \& (\langle y \rangle_I ; y') \prec \langle x \rangle_I ; (\langle y \rangle_I ; y')$$

$$+ \langle y \rangle_I ; y' \& \langle x \rangle_I$$

$$\langle x \rangle_I \& \langle y \rangle_I \prec \langle x \rangle_I ; \langle y \rangle_I$$

$$+ \langle y \rangle_I ; \langle x \rangle_I$$