Advanced Mechanisms for Service Combination and Transactions

António Ravara[1]

Dep of Informatics, FCT, New Univ of Lisbon

Milan, November 24, 2009

[1] with Carla Ferreira, Ivan Lanese, and Hugo T. Vieira

Long Running Transactions

What are they

- Computer activities that may last long periods of time.
- Common on systems composed by loosely coupled components, like service-oriented systems.

What can go wrong

Unexpected events may cause premature termination before the completion of the transaction.

- System failures like unreachability or time-out.
- A partner is not willing to participate anymore in the transaction.
- ...

Incomplete Transactions

How to handle premature termination

- Not feasible to lock (non-local) resources, thus, these transactions do not enjoy some of the usual ACID properties.
- Necessary to foresee special activities to *recover* from partial transaction execution.
- Purpose: lead the system to a sound state.

Recovery mechanisms

- Exception-handling: uses primitives to try-catch and throw failure signals.
- Compensation-handling: uses primitives to install and activate dedicated activities.

Contents of the Chapter

Linguistic primitives to deal with transaction failure

Main features under inspection are the mechanisms to deal with:

- failures: *exceptions* or *compensations*;
- non-interruptable units of execution: *protection operator*;
- nested computations: *nested transactions* and *nested failures*.

Sections

1. Linguistic primitives for exception and compensation handling.
2. Applications of the mechanisms in the context of SOC.
3. Models to reason about the mechanisms.

Table of Contents

Comparing two primitives

Interrupt versus try-catch

- $P \triangle Q$ executes P until Q executes its first action; when Q starts executing, the process P is interrupted
- $\texttt{try}\, P\, \texttt{catch}\, Q$ operator executes P, but if P performs a *throw* action it is interrupted and Q is executed instead

Failure management by example

- Failures managed externally; interruption not atomic
  $\texttt{PAY}; \texttt{if}\ !res\ \texttt{then}\ throw\ \texttt{else}\ 0; \ldots \triangle (f.\ \texttt{manageFault}) \mid \overline{throw}.\overline{f}$
- Decision to interrupt the execution of P is taken inside P itself
  $\texttt{try}\ \texttt{PAY}; \texttt{if}\ !res\ \texttt{then}\ \texttt{throw}\ \texttt{else}\ 0; \ldots\ \texttt{catch}\ \texttt{manageFault}$

Expressiveness

Summary of results

	interrupt	try-catch
repl	$CCS_{!}^{\triangle}$ exist termination undec univ termination decid	$CCS_{!}^{\mathrm{tc}}$ exist termination undec univ termination decid
rec	$CCS_{rec}^{\triangle}$ exist termination undec univ termination decid	CCS_{rec}^{tc} exist termination undec univ termination undec

Expressiveness

Discrimination results

- **Interruption cannot be encoded using only communication primitives.** In CCS without restriction, existential termination is decidable while it is undecidable with either interrupt or try-catch
- **try-catch mechanism cannot be encoded using communication primitives and the interrupt operator.** With recursion universal termination is decidable in the presence of the interrupt operator, while this is not the case for try-catch

Compensation policies

Overview

	compensation definition	nested vs non-nested	protection operator
πt [BLZ03]	static	nested	no
c-join [BMM04]	static	nested	no
webπ [LZ05]	static	non-nested	implementable
webπ_∞ [ML06]	static	non-nested	implementable
dcπ [VFR08]	parallel	nested	yes
CaSPiS [BBDL08]	static	nested	no
CC [VCS08]	static	nested	no
COWS [LPT07]	static	nested	yes
SOCK [GLMZ08]	dynamic	nested	implementable

Table: Features of calculi and languages with compensation handling.

Static compensations

webπ_∞: workunit construct

- Workunit $\langle\!| P \;;\; Q |\!\rangle_t$ executes P until receiving message $\overline{t}$; then, P is killed and compensation Q is executed
- $\langle\!|$ PAY.if $!res$ then $\overline{t}$ else 0.... ; manageFault$|\!\rangle_t$
- Weak asynchronous bisimilarity characterises weak barbed congruence
- Handlers reducibility:

$$\langle\!| P \;;\; Q |\!\rangle_x = (x'x'')(\langle\!| P \;;\; \overline{x'} |\!\rangle_x \mid \langle\!| x'.Q \;;\; \mathbf{0} |\!\rangle_{x''})$$

for each $x', x'' \notin \mathrm{fn}(P) \cup \mathrm{fn}(Q), x' \neq x'' \neq x$

Dynamic compensations

Parallel recovery: $\mathtt{dc}\pi$

- Input and compensation update form a unique atomic primitive

$$payConf(\vec{x})\%\overline{Annul}\langle\vec{x}\rangle.Q$$

- Message $\overline{payConf}\langle\vec{v}\rangle$ installs in the nearest enclosing scope a new compensation item $\overline{Annul}\langle\vec{v}\rangle$ and continues as $Q\{\vec{v}/\vec{x}\}$
- When a scope is killed, all the installed compensation items are executed in parallel

Dynamic compensations

General recovery policies: backward, parallel, or forward

- Compensable processes provide
  1. a scope construct $t[P, Q]$
  2. a compensation update primitive $\mathrm{inst}\lfloor \lambda X.Q' \rfloor.R$
- Parallel recovery: $Q' = Q'' \mid X$ where X does not occur in Q''
- Backward recovery: $\lambda X.(\mathit{finished})(Q' \mid \mathit{finished}.X)$
  The Q' signals its termination with an output on the private channel *finished*
- Forward recovery: the compensation can be deleted by installing $\lambda X.\mathbf{0}$, or replaced with a new compensation by installing $\lambda X.\mathit{NewComp}$ where *NewComp* does not contain X

Dynamic compensations

Example of backward recovery

$$t[PAY_1.\,\mathsf{inst}\lfloor \lambda X.ANNUL_1.X \rfloor.\ldots.PAY_n.\,\mathsf{inst}\lfloor \lambda X.ANNUL_n.X \rfloor.\, CHECK.\ \texttt{if}\ check = \texttt{ok}\ \texttt{then}\ \mathsf{inst}\lfloor \lambda X.\mathbf{0} \rfloor\ \texttt{else}\ \bar{t}, \mathbf{0}]$$

If something goes wrong in one of the payments, all are annulled. At the end a final check is performed, and if it succeeds then annul is no more possible.

Dynamic compensations

Expressiveness Results

- Parallel recovery is encodable into static recovery, preserving weak bisimilarity
- No "good" encoding of backward or forward recovery to static recovery exists

Implementing SAGAs in SOCK

SAGAs [BMM05] are (sequential or parallel) compositions of basic compensable activities

Encoding

- Activities as Services, invoked using the request-response interaction pattern
- Failures of activities generate faults, handled by the automatic fault notifcation mechanism of SOCK
- Abortion of a SAGA is managed by using SOCK fault and compensation handlers
- Encoding proved correct.

Reasoning about structured compensating transactions

A general model

- To reason about compensations in an abstract way, independently from a particular language implementation [CFV08]
- Compensating CSP (cCSP) enjoys of fundamental properties expected in any compensation model, namely atomicity of transactions
- There is a correct embedding of cCSP transactions in the Conversation Calculus, since it induces a stateful model of compensating transactions

Reasoning about structured compensating transactions

Compensation Model

A compensation model is a pair $(\mathcal{S}, \mathcal{D})$ where $\mathcal{S}$ gives its static structure and $\mathcal{D}$ gives its dynamic structure

- The static structure $\mathcal{S} = (S, \mid, \#, \bowtie)$ is defined such that:
  - S is a set of (abstract) states
  - $\mid$ is a partial composition operation on states
  - $\#$ is an apartness relation on states
  - $\bowtie$ is an equivalence relation on S
- The dynamic structure $\mathcal{D} = (\Sigma, \xrightarrow{a})$ is defined such that:
  - Σ is a set of primitive actions
  - $\xrightarrow{\text{a}}$ is a labeled (by elements of Σ) transition system between states.

Reasoning about structured compensating transactions

Results

- The behavior of transactions implemented over $\bowtie$-consistent compensable programs approximate atomicity: a transaction either aborts (*throw*) doing “nothing”, or ($\oplus$) terminates successfully after executing all of its forward actions (R^+)
- There is a correct encoding of Structured Compensating Transactions in CC

M. Boreale, R. Bruni, R. De Nicola, and M. Loreti.
Sessions and pipelines for structured service programming.
In *Proc. of FMOODS'08*, volume 5051 of *LNCS*, pages 19–38. Springer, 2008.

L. Bocchi, C. Laneve, and G. Zavattaro.
A calculus for long-running transactions.
In *Proc. of FMOODS'03*, volume 2884 of *LNCS*, pages 124–138. Springer, 2003.

R. Bruni, H. Melgratti, and U. Montanari.
Nested commits for mobile calculi: Extending join.
In *Proc. of IFIP TCS'04*, pages 563–576. Kluwer, 2004.

R. Bruni, H. Melgratti, and U. Montanari.
Theoretical foundations for compensations in flow composition languages.
In *Proc. of POPL '05*, pages 209–220. ACM Press, 2005.

L. Caires, C. Ferreira, and H.T. Vieira.
A process calculus analysis of compensations.
In *Proc. of TGC'08*, volume 5474 of *LNCS*, pages 87–103. Springer, 2008.

C. Guidi, I. Lanese, F. Montesi, and G. Zavattaro.
On the interplay between fault handling and request-response service invocations.
In *Proc. of ACSD'08*, pages 190–199. IEEE Computer Society Press, 2008.

A. Lapadula, R. Pugliese, and F. Tiezzi.
A calculus for orchestration of web services.
In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.

C. Laneve and G. Zavattaro.
Foundations of web transactions.

In *Proc. of FoSSaCS'05*, volume 3441 of *LNCS*, pages 282–298. Springer, 2005.

M. Mazzara and I. Lanese.
Towards a unifying theory for web services composition.
In *Proc. of WS-FM'06*, volume 4184 of *LNCS*, pages 257–272. Springer, 2006.

H.T. Vieira, L. Caires, and J.C. Seco.
The conversation calculus: A model of service-oriented computation.
In *Proc. of ESOP'08*, volume 4960 of *LNCS*, pages 269–283. Springer, 2008.

C. Vaz, C. Ferreira, and A. Ravara.
Dynamic recovering of long running transactions.
In *Proc. of TGC'08*, volume 5474 of *LNCS*, pages 201–215. Springer, 2008.