TECHNICAL REPORT

Report No. CS2011-01
Date: February 2011

A Compensating Transaction Example in Twelve Notations

Christian Colombo
Gordon J. Pace

Department of Computer Science
University of Malta
Msida MSD 06
MALTA

Tel: +356-2340 2519
Fax: +356-2132 0539
http://www.cs.um.edu.mt

A Compensating Transaction Example in Twelve Notations

Christian Colombo
Department of Computer Science
University of Malta
Msida, Malta
christian.colombo@um.edu.mt

Gordon J. Pace
Department of Computer Science
University of Malta
Msida, Malta
gordon.pace@um.edu.mt

Abstract: *The scenario of business computer systems changed with the advent of cross-entity computer interactions: computer systems no longer had the limited role of storing and processing data, but became themselves the players which actuated real-life actions. These advancements rendered the traditional transaction mechanism insufficient to deal with these new complexities of longer multi-party transactions.*
The concept of compensations *has long been suggested as a solution, providing the possibility of executing "counter"-actions which semantically undo previously completed actions in case a transaction fails.*
There are numerous design options related to compensations particularly when deciding the strategy of ordering compensating actions. Along the years, various models which include compensations have emerged, each tackling in its own way these options.
In this work, we review a number of notations which handle compensations by going through their syntax and semantics — highlighting the distinguishing features — and encoding a typical compensating transaction example in terms of each of these notations.

A Compensating Transaction Example in Twelve Notations*

Christian Colombo
Department of Computer Science
University of Malta
Msida, Malta
christian.colombo@um.edu.mt

Gordon J. Pace
Department of Computer Science
University of Malta
Msida, Malta
gordon.pace@um.edu.mt

Abstract: *The scenario of business computer systems changed with the advent of cross-entity computer interactions: computer systems no longer had the limited role of storing and processing data, but became themselves the players which actuated real-life actions. These advancements rendered the traditional transaction mechanism insufficient to deal with these new complexities of longer multi-party transactions.*

The concept of compensations *has long been suggested as a solution, providing the possibility of executing "counter"-actions which semantically undo previously completed actions in case a transaction fails.*

There are numerous design options related to compensations particularly when deciding the strategy of ordering compensating actions. Along the years, various models which include compensations have emerged, each tackling in its own way these options.

In this work, we review a number of notations which handle compensations by going through their syntax and semantics — highlighting the distinguishing features — and encoding a typical compensating transaction example in terms of each of these notations.

*The research work disclosed in this publication is partially funded by the Malta National Research and Innovation (R&I) Programme 2008 project number 052.

1 Introduction

Up to a few decades ago, the main aim of computer systems in businesses was to store records of information and be able to process them in an efficient way. In general, computer systems of an entity did not interact with another's. Therefore, the main challenge was to keep data safe and consistent, and to handle concurrency issues if the system was accessed from several points within the entity. To this end, the concept of a transaction has been widely used, ensuring that a change in the data is either complete or undetectable.

The scenario changed with the advent of online interactions among computer systems of several entities. At this point, computer systems no longer had the limited role of storing and processing data, but became themselves the players which actuated real-life actions. For example, originally, a travel agent system would have required a human operator which contacted an airline to book the flight, a hotel to book the accommodation, and a taxi operator to book the transport. When successful, the operator would input all the information in the travel agency computer system. Subsequently, the customer would be told about the success of the transaction and pays for the service. The scenario now is different: an online travel agency will allow the client to submit the flight, hotel and transport details and then attempt to automatically actuate all the bookings, charging the customer's bank account if successful. This new model, thus, brought about the following changes: (i) since interactions involved several entities, transactions began to take long to complete; and (ii) since computer transactions not only reported but actuated transaction actions, the possibility of failure of such real-life actions necessitated the option of semantically reversing (real-life) actions — real-life actions cannot be simply undone (or erased from a database) but rather reversed by a compensating action.

These changes rendered the traditional transaction mechanism insufficient to deal with the new complexities. Particularly, since resource locking is a fundamental mechanism for the implementation of traditional transactions, these were not appropriate for handling large volumes of transactions which take long to complete. As a solution, the concept of compensation has been suggested so that rather than waiting for the whole transaction to complete, any completed part is considered successful with the possibility that if some other part fails later on, the affected parts can be compensated for. Taking again the example of the travel agency, if all bookings succeed but the payment by the customer fails, then the travel agency might decide to cancel the previously completed bookings. Note that compensating for an activity does not necessarily mean that the activity is undone in exactly the reverse way. For example, as a compensation for bookings, one might not only need to cancel each booking but also inform the client and keep some form of record for future reference.

Compensation is a very rich construct since, in contrast to the usual sequential or parallel programming, compensation allows one to execute a number of steps based on the history of the execution. There are numerous design options related to compensation particularly when deciding the strategy of ordering the compensating actions. Along the years, various models which include compensations have emerged, each tackling in its own way the options which such a mechanism poses.

There were several attempts of formalising the notion of compensation, mainly motivated by the need of clear semantics and the possibility of applying verification techniques. These efforts resulted in numerous formalisms and notations with notable differences in the way they handle compensations. In this report, we will analyse the compensation mechanism of each formalism in depth, going into the actual formal semantics where necessary. The formalisms are organised depending whether they assume a centralised coordination mechanism — orchestration approaches (Section 3), or whether they assume processes collaborate together in a decentralised fashion — choreography approach (Section 4). Next, we dedicate a section to BPEL (Section 5), which although an orchestration approach is treated separately due to the amount of worked revolving around it. Finally, we compare the orchestration and choreography approaches in Section 6). To further help in the illustration of these formalisms, we shall present a simple yet non-trivial example (Section 2) which will subsequently be represented in the different notations, enabling us to explain and compare them.

2 An Online Bookshop Transaction Example

In order to compare the compensation notations presented in the following sections, we present in this section a substantial example consisting of an online bookshop transaction. To help delineate the differences among the notations, we include in the example the main features which one normally encounters in long running transactions: sequential and parallel activities, alternative forwarding[1], programmable compensation[2] and speculative choice[3]. In the following two subsections we first give a textual description of the example and then represent the example diagrammatically for more clarity.

2.1 Textual Description

An online bookshop receives an order from a client and the shop initiates an order handling procedure. First, it attempts to deduct the respective stock levels (implicitly checking that there is sufficient stock available). If this is successful, the client is redirected to a marketing promotion offer. The client is allowed to refute the offer, in which case another one is presented. Subsequently, whether or not an offer is taken up, two processes are started in parallel: the packing of the ordered books and the money transfer for payment. These processes are handled by the bookshop and the bank respectively. If both of these activities succeed, then two couriers are contacted (concurrently) and the first one accepting to deliver the order is chosen while the other booking attempt is cancelled.

If a failure occurs in any of the above mentioned processes, the previously completed activities must be compensated. Therefore, if the courier booking fails, the money taken from the customer's account is refunded and the books are unpacked, the offer is withdrawn, the stock is increased to its original level and an email is sent to the customer indicating the failure of the order delivery. Similarly, if the money transfer fails, the packing is undone (or cancelled if it has not yet completed), the offer is

[1]The alternative forwarding mechanism provides an alternative way of reaching a goal such that if an activity fails, but successfully compensates, then the alternative is tried out. If the alternative succeeds, then the transaction does not continue with compensation but rather continues its normal execution.

[2]A programmable compensation refers to the replacement of a fine-grained compensation with one which is coarse grained. Put differently, after the completion of a number of activities (each with possibly a compensation activity), programmable compensation allows the installation of a single compensation which replaces the various installed compensation activities.

[3]A speculative choice starts the execution of two processes in parallel, stopping one if the other succeeds. If both succeed concurrently, then an arbitrary one is compensated for, while it is considered as failed if both processes fail.

withdrawn, stock is updated and an email is sent to the client. Thus, the activities executed in case of a fault depend on the point where the fault occurs. Note that irrespective of the order in which the activities were carried out, their compensations can be carried out in parallel since there is no dependency among the compensations.

Finally, if the transaction and its compensation both fail, an operator is notified so that the failure can be handled manually. It is assumed that any of the mentioned activities may fail.

2.2 Illustrative Description

After the informal description of the previous subsection, we now give the diagrammatic representation of the example making use of the following abbreviations:

Transaction	the main book ordering transaction
Order	accepting an order and reducing the stock
Pack	packing the order
Credit	charging the customer's credit card
$Courier_i$	booking the first ($i = 1$) or second ($i = 2$) courier
$Cancel_i$	cancelling the first or second courier
Refund	refunding the money paid
Unpack	unpacking the order
ReStock	incrementing stocks to their previous levels
Email	sending an email to the customer
$Offer_i$	presenting the first or second offer
Withdraw	withdrawing an offer
Operator	manual intervention
Shop	the section which receives orders from the client
Packing	the section responsible for packing the order
Bank	the section responsible for crediting the client's bank account
Offers	the section responsible for the offers
Couriers	the section responsible for booking couriers

As shown in Figure 1, we use the Business Process Modelling Notation (BPMN) [bpm08] to represent our example. BPMN is defined by the Object Management Group (OMG) and provides pictorial business process modelling notation supporting compensations. There have been various attempts of formalising BPMN ([DDO07, DGHW07, WG08, Tak08, DDDGB08]). However, we refrain from going deeply into BPMN's semantics since the compensation concepts are similar to BPEL's (see Section 5 regarding BPEL and [bpm08] for a mapping from BPMN to BPEL).

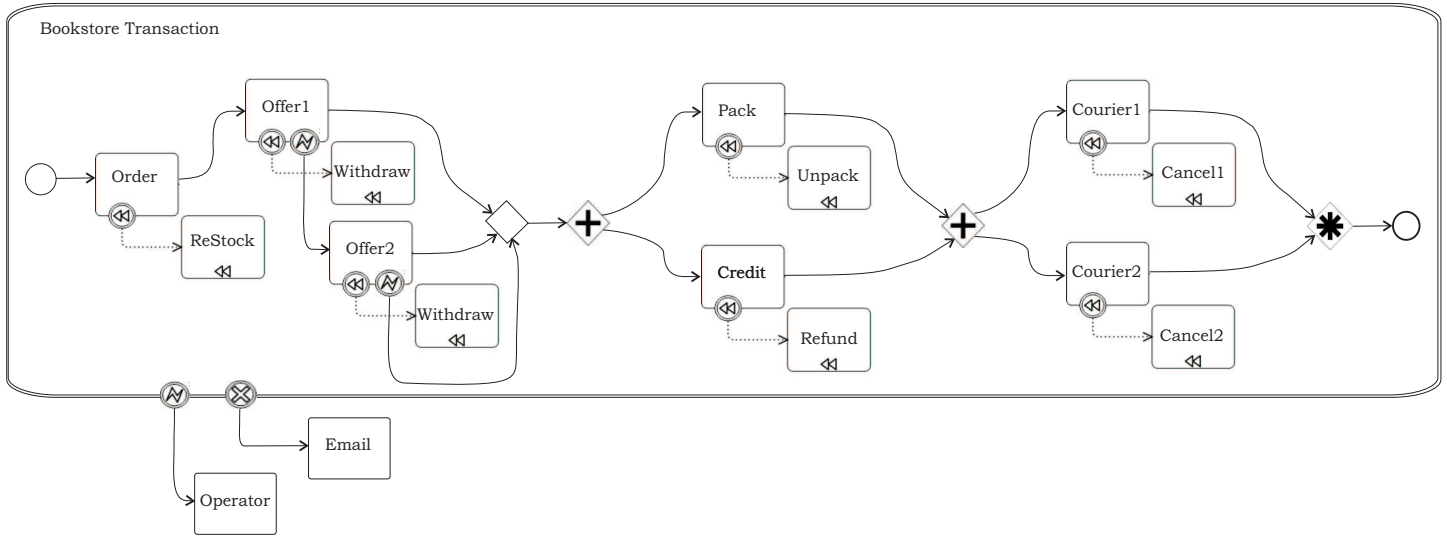

Figure 1: A BPMN representation of the example.

Informally, the symbols used in Figure 1 have the following meanings: a circle represents the starting point of execution when the parent transaction obtains control; a bold circle represents the end of execution; a diamond (with not symbol inside) represents a simple merge of control flows (exactly one input token is expected); a diamond with a plus symbol inside represents a merge of parallel control flows; a diamond with an asterisk symbol inside represents a complex decision (among a number of alternatives), allowing only the first incoming token to go through; a double circle with the "rewind" symbol signifies that the activity has an associated compensation; a double circle with the "lightning" symbol signifies that the activity has an associated exception handler; a double circle with the times symbol signifies that if the transaction is cancelled, then an attached activity is to be executed.

Note that due to space limitations we do not explicitly show compensation requests in the diagram. However, we assume that each activity can trigger an exception which in turn triggers compensation.

In the three sections which follow, we refer to this example and encode it using the various notations presented.

3 Orchestration Approaches

In this section, we present a number of orchestration approaches which are able to handle compensations. Orchestration approaches, sometimes referred to as flow-based approaches, assume a centralised coordination mechanism which manages the execution of the activities involved. This means that the activities themselves need not be aware of that they are part of a composition of activities. The consequence is that processes do not need to interact with each other and notations supporting such approaches provide structured operators enabling one to build complex activities out of basic ones.

In what follows we review a number of orchestration approaches, briefly going through their syntax and semantics, enabling us to highlight the main features of each approach and explain the respective example encoding. Note that BPEL is also an orchestration approach but we opt to dedicate a whole section (Section 5 due to the considerable number of works which revolve around it.

3.1 Sagas

Sagas [BMM05] is a flow composition language based on the idea of sagas introduced by Garcia-Molina and Salem [GMS87] where, in essence, a sequential saga is a long-lived transaction (i.e. a transaction which takes long to complete, abbreviated as LLT) composed of a sequence of activities which either succeed in totality (all activities succeed) or else the completed activities should be compensated (giving the illusion that none of the LLT was successful). Starting from this basic structure, Bruni et al. [BMM05] go on to provide a hierarchy of extensions including parallel composition, nesting of transactions, programmable compensations and various other highly expressive features.

Syntax

Given an infinite set of basic activities $\mathcal{A}$ (each of which may either succeed or fail, but not partially succeed) ranged over by A and B, steps (ranged over by X), processes (ranged over by P) and sagas (ranged over by X) are defined as follows:

$$\begin{array}{rcll} X & ::= & 0 \mid A \mid A \div B & \text{step} \\ P & ::= & X \mid P\,;P \mid P|P \mid P \boxplus P \mid S & \text{process} \\ S & ::= & \{\!|P|\!\} \mid S \div P \mid \textbf{try } S \textbf{ with } P \mid \textbf{try } S \textbf{ or } P & \text{saga} \end{array}$$

A step X may either be the inert activity 0; an activity A; or an activity A with a compensation B, $A \div B$. Note that B is an activity and cannot itself have compensa-

tions. A process is either a step; a sequence or a parallel composition of processes; a speculative choice $P \boxplus P$; or a saga (for nesting transactions). Finally, a saga is either a process enclosed within a scope written as $\{\!|P|\!\}$; a saga S with programmable compensation P, written as $S \div P$ (introducing the possibility of a compensation having compensations); a saga with exception handling, represented by **try** S **with** P, such that if S terminates abnormally, P is executed; or a saga with alternative forwarding, **try** S **or** P, such that if S fails but successfully aborts, P is tried out such that if P succeeds, the transaction continues forward.

Semantics

The semantics of Sagas is given as big-step operational semantics and the success or failure of an action depends on a context Γ, abstracting away from explicitly programming failure. For example, the rule $\Gamma \vdash \langle P, \beta \rangle \xrightarrow{\alpha} \langle \square, \beta' \rangle$ signifies that under context Γ, transaction P evolves to (terminates as) $\square$ if the actual flow of control is α. Furthermore, given that the starting compensation is β, the compensation evolves to β'. Note that an activity is considered to be atomic; i.e. either it succeeds fully, or it leaves no trace of its execution. A transaction may terminate in three different ways: $\square \in \{\boxdot, \boxtimes, \circledast\}$ where $\boxdot$ signifies that a transaction terminated successfully, $\boxtimes$ signifies that the transaction has been correctly aborted; i.e. a failure occurred but the transaction have successfully compensated, while $\circledast$ signifies an abnormally terminated transaction; i.e. one which failed and whose compensation has failed as well.

First, we consider what happens upon a successful completion of a step. Recall that the point of completion is the point when an activity becomes compensable. Applying this principle, upon successful completion of a step $A \div B$, B is added as a prefix to the currently accumulated compensation. The rule is given as follows:

$$A \mapsto \boxdot, \Gamma \vdash \langle A \div B, \beta \rangle \xrightarrow{A} \langle \boxdot, B \,;\, \beta \rangle$$

Specifically, note that $A \div B$ evolves to $\boxdot$, and that β evolves to $B \,;\, \beta$. Next, we focus on the semantic rules handling the compensation mechanism of Sagas, starting with the following:

$$\frac{\Gamma \vdash \langle \beta, 0 \rangle \xrightarrow{\alpha} \langle \boxdot, 0 \rangle}{A \mapsto \boxtimes, \Gamma \vdash \langle A \div B, \beta \rangle \xrightarrow{\alpha} \langle \boxtimes, 0 \rangle} \qquad \frac{\Gamma \vdash \langle \beta, 0 \rangle \xrightarrow{\alpha} \langle \boxtimes, 0 \rangle}{A \mapsto \boxtimes, \Gamma \vdash \langle A \div B, \beta \rangle \xrightarrow{\alpha} \langle \circledast, 0 \rangle}$$

The first rule handles the case where compensation succeeds — β progresses to $\boxdot$ — leading to an aborted transaction when A fails ($A \mapsto \boxtimes$), i.e. $A \div B$ evolves to $\boxtimes$. The

second rule states what happens if a failure occurs ($A \mapsto \boxtimes$) and the compensation fails as well (β progresses to $\boxtimes$). This leads to an abnormal termination, i.e. $A \div B$ evolves to $\circledast$. An interesting aspect of Sagas's compensation mechanism is that the parent of a successfully aborted saga is unaware that one of its child sagas have aborted. This means that abortion does not in any way affect the parent if it is successful. The rule which hides abortion is defined as follows:

$$\frac{\Gamma \vdash \langle P, 0\rangle \xrightarrow{\alpha} \langle \boxtimes, 0\rangle}{\Gamma \vdash \langle \{\!|P|\!\}, \beta\rangle \xrightarrow{\alpha} \langle \boxdot, \beta\rangle}$$

Note how $\{\!|P|\!\}$ evolves to $\boxdot$, even though P evolves to $\boxtimes$. The situation is completely different when the compensation fails. In such a case the parent is fully aware that one of its sub-sagas have failed. The consequences that a failed child brings about are explained further below.

Next, we consider sequential and parallel composition of processes. Sequential composition is quite straight forward such that if the first process fails the sequential composition is considered a failure (Not even attempting the second one). Otherwise, the compensation of the first process is remembered so that if the second process fails, the former is compensated. However, the case of parallel composition is more complex. If a parallel composition succeeds, the compensation installed is the parallel composition of the compensations of both branches. The rule is as follows:

$$\frac{\Gamma \vdash \langle P, 0\rangle \xrightarrow{\alpha} \langle \boxdot, \beta'\rangle \quad \Gamma \vdash \langle Q, 0\rangle \xrightarrow{\alpha'} \langle \boxdot, \beta''\rangle}{\Gamma \vdash \langle P|Q, \beta\rangle \xrightarrow{\alpha|\alpha'} \langle \boxdot, \beta'|\beta''\,;\beta\rangle}$$

In particular, note that the compensations of the parallel processes P and Q, β' and β'' (respectively) are added as a parallel composition in front of the accumulated compensation β. In the other case where one of the parallel branches fails, it is usually desirable that the other processes running in parallel are interrupted so that compensation starts immediately. Such interruption is referred to as *forced termination*. If the compensation resulting from a forced termination is successful, it is symbolised as $\overline{\boxtimes}$; otherwise, the symbol $\overline{\circledast}$ is used. If the compensations of both parallel branches succeed, then the outcome of the overall parallel composition is considered as a successful abortion. Yet, there is also the possibility that one of the parallel branches fails (i.e. resulting in either $\circledast$ or $\overline{\circledast}$). If this is the case, the other branch (unless it has already compensated or failed) is forced to terminate and execute its compensations. However, the forced compensation might also fail resulting in the four possible outcomes $\boxtimes$, $\circledast$, $\overline{\boxtimes}$ and $\overline{\circledast}$. The rule handling this scenario is as follows:

$$\frac{\Gamma \vdash \langle P, 0\rangle \xrightarrow{\alpha} \langle \sigma_1, 0\rangle \quad \Gamma \vdash \langle Q, 0\rangle \xrightarrow{\alpha} \langle \sigma_2, 0\rangle}{\Gamma \vdash \langle P|Q, \beta\rangle \xrightarrow{\alpha|\alpha'} \langle \sigma_1 \wedge \sigma_2, 0\rangle} \quad \begin{cases} \sigma_1 \in \{\circledast, \overline{\circledast}\} \\ \sigma_2 \in \{\boxtimes, \circledast, \overline{\boxtimes}, \overline{\circledast}\} \end{cases}$$

The conjunction ($\wedge$) of transaction terminating symbols has the aim of giving an overall verdict of the parallel composition such that failure takes over abortion (eg. $\boxtimes \wedge \boxtimes = \boxtimes$ but $\circledast \wedge \boxtimes = \circledast$), while non-forced termination takes over forced termination. Thus, since at least one of the parallel branches is a non-forced termination, the final outcome of the transaction can never be a forced termination. This rule also explains how failure is propagated within a saga, such that if a child of the saga fails, then the whole saga fails by propagating the failure to any parallel branches.

Example

The example of the bookstore given in Sagas is as follows:

$$\begin{aligned} \mathit{Trasaction} \stackrel{\text{def}}{=}\ & \textbf{try}\ \{\!\![\\ & \quad (\mathit{Order} \div \mathit{ReStock})\,;(0 \div \mathit{Email})\,; \\ & \quad (\{\!\![\textbf{try}\ \mathit{Offer}_1\ \textbf{or}\ \mathit{Offer}_2]\!\!\} \div \mathit{Withdraw})\,; \\ & \quad ((\mathit{Pack} \div \mathit{Unpack})|(\mathit{Credit} \div \mathit{Refund}))\,; \\ & \quad ((\mathit{Courier}_1 \div \mathit{Cancel}_1) \boxplus (\mathit{Courier}_2 \div \mathit{Cancel}_2)) \\ &]\!\!]\ \}\ \textbf{with}\ \mathit{Operator} \end{aligned}$$

Since Sagas employs big step semantics, the actions (*Order*, *ReStock*, etc) are not decomposed further. The possibility of these succeeding or failing is handled by the semantics of Sagas through a context Γ. A limitation of Sagas is that it does not allow a compensation to be a process. For this reason, the sending of an email as a compensation for a successful order has been encoded as the compensation of the inert process ($0 \div \mathit{Email}$). By so doing, we have not strictly kept to the specification because the stock update and the sending of the email are supposed to be done in parallel. The alternative would have been to amalgamate both activities as a single basic activity.

In this example, we have also deviated from the original specification of the example in two main aspects: (i) the compensations cannot be executed in parallel because in Sagas the compensations of sequential processes are executed in the reverse order of the original execution; (ii) the interaction among the various entities involved in the transaction cannot be modelled (eg. we cannot model the client communicating with the bookstore). The latter is a limitation of all flow composition languages.

3.2 Compensating CSP

Compensating CSP (cCSP) [BHF04, BR05] is an extension to CSP [Hoa85] with the aim of providing support for LLTs. In cCSP, all basic activities succeed and failures are explicitly programmed using a special *THROW* activity which always fails. Although cCSP executable semantics has been given [BR05], the cCSP semantics has originally been given in terms of traces [BHF04]. In this overview, we use the latter to explain the calculus.

Syntax

cCSP has two kinds of processes: standard processes and compensable processes. Standard processes are unaware of compensations. When two standard processes are glued together, one as forward behaviour and another backward behaviour (using the $\div$ operator), they form a compensable process. Given a set of atomic actions ranged over by A, the syntax of cCSP standard processes (ranged over by P) and compensable processes (ranged over by PP) is given as follows:

$$
\begin{array}{rcll}
P & ::= & A \mid P\,;P \mid P \,\Box\, P \mid P \,||\, P \mid \mathit{SKIP} & \text{standard process} \\
 & \mid & \mathit{THROW} \mid \mathit{YIELD} \mid P \rhd P \mid [PP] & \\
PP & ::= & P \div P \mid PP\,;PP \mid PP \,\Box\, PP \mid PP \,||\, PP & \text{compensable process} \\
 & \mid & \mathit{SKIPP} \mid \mathit{THROWW} \mid \mathit{YIELDD} &
\end{array}
$$

A standard process P can be an atomic action A; a sequence of two processes, written as $P\,;P$; a choice between two processes, $P \,\Box\, P$; a parallel composition, $P \,||\, P$; the process *SKIP* which represents the inert process; *THROW* representing a process which always fails; *YIELD* which signifies the possibility of yielding (see below) to an interrupt; a process having another process as an interrupt handler, written as $P \rhd P$; or a transaction block $[PP]$.

A compensable process is either a pair of standard processes $P \div P$ where the latter is a compensation of the former; a sequence, choice or parallel composition of two compensating processes; or *SKIPP*, *THROWW*, or *YIELDD* — the counterparts of *SKIP*, *THROW*, *YIELD* for compensable processes.

Some interesting things to point out directly from the syntax are that: (i) compensations cannot have compensations; (ii) a transaction block automatically discards its compensation (concluded from the fact that $[PP]$ is a standard process); (iii) and nested transactions are supported. The fact that $[PP]$ is a standard process implies that a compensation can be attached to a transaction block as soon as it completes (programmable compensation).

Semantics

The semantics of a cCSP standard process is given in terms of a set of possible traces describing possible behaviours of the process. In the case of a compensable process the semantics is given as a set of pairs of traces (since a compensation may be installed). A trace is a sequence of basic actions ending in either $\langle\checkmark\rangle$ — signifying success, $\langle!\rangle$ — signifying failure, and $\langle?\rangle$ signifying a yield.

The purpose of the *YIELD* operator is to allow processes running in parallel to "yield" to forced termination if one of them fails. In other words, yields mark points within a process indicating that at those points the process allows itself to be interrupted. Consider the following equation:

$$THROW \,||\, (YIELD\,;P) = THROW \,\Box\, P\,;THROW$$

Note that if the *THROW* process occurs before the start of the second process, the latter does not even start execution because it allows itself to be interrupted by the exception. This explains the possibility of a solitary *THROW* in the right hand side of the equation. However, since parallel composition is defined as the interleaving of the activities, P can take place before *THROW* (hence the other possibility). Another observation is that unlike various other formalisms whose forced termination stops the other parallel processes instantly, in this case another process is only terminated when it yields to the interrupt.

Apart from providing the compensation mechanism, cCSP also supports exception handling. In fact, compensations are triggered in the case of an unhandled exception (or a failed exception handler). Consider two traces p and q, and two processes P and Q each representing a set of traces (corresponding to their behaviour). The semantics of Q as an exception handler of P, $P \rhd Q$, is given as:

$$\begin{array}{rcl} p\langle!\rangle \rhd q & = & pq \\ p\langle\omega\rangle \rhd q & = & p\langle\omega\rangle \quad (\omega \in \{\checkmark, ?\}) \\ P \rhd Q & = & \{p \rhd q \mid p \in P \wedge q \in Q\} \end{array}$$

Note that if a trace ends with failure ($\langle!\rangle$) then the exception handler is executed — hence the resultant pq. Otherwise, q is simply ignored (in the second case). Finally, the semantics of $P \rhd Q$ is the application of $\rhd$ on each pair of possible traces for each process.

Next, we consider the installation of compensations. The installation of compensation for sequential processes is installed as a prefix of the previously accumulated compensations (resulting in the reverse order in which the activities were originally

executed), while the compensation of parallel processes is installed as the parallel composition of the respective compensations.

Subsequently, we consider the execution of compensations. Upon the completion of a transaction block, a decision is taken regarding whether the accumulated compensation is to be activated or discarded. Consider the following equation giving semantics for a transaction block $[PP]$:

$$[PP] = \{pp' \mid (p\langle!\rangle, p') \in PP\} \cup \{p\langle\checkmark\rangle \mid (p\langle\checkmark\rangle, p') \in PP\}$$

If the forward behaviour of PP terminates with $\langle!\rangle$ (the first part of the union), then the installed compensation p' is executed. Otherwise, the compensation is simply discarded (the second part of the union). Note that a successfully compensated transaction is considered a successful transaction by the parent transaction (since the termination of p' is preserved in pp'). Another interesting observation is that the above equation does not consider the possibility of a trace ending in $\langle?\rangle$. The reason is that, forced termination (recall that a trace ending with yield ($\langle?\rangle$) signifies a forced termination of a parallel branch) is not allowed to surface up to the level of a transaction block. An interrupted parallel branch is in fact handled by the following equations where $\omega, \omega' \in \{\checkmark, ?, !\}$, the & operator combines possible terminations such that abnormal termination takes over forced (yielding) termination, and forced termination takes over successful termination (eg. ! & ? = !), and ||| returns a set of all possible interleavings of two traces.

$$\begin{aligned} p\langle\omega\rangle \,||\, q\langle\omega'\rangle &= \{r\langle\omega\&\omega'\rangle \mid r \in (p \,|||\, q)\} \\ P \,||\, Q &= \{r \mid r \in (p \,||\, q) \wedge p \in P \wedge q \in Q\} \end{aligned}$$

Firstly, the parallel composition of two traces is thus the sets of all interleavings of both traces with the combination of their termination as the resulting termination. Secondly, the parallel composition of two compensable processes P and Q is the parallel composition of all pairs of possible traces of P and Q.

Subsequently, we consider the choice operator ($\Box$) which is crucial in cCSP because it models the uncertainty which is usually associated with third party service invocation. (Recall that all basic activities in cCSP succeed.) Therefore, the choice operator is indispensable for modelling the possible failure of certain activities. Defined in terms of choice, cCSP also provides speculative choice ($\boxtimes$) such that two functionally equivalent processes are started in parallel and the first one to commit causes the second one to abort.

Example

The example of the bookstore encoded in cCSP is given as follows:

$$
\begin{array}{rcl}
\mathit{Order} & \stackrel{\text{def}}{=} & \mathit{Order}' \,\Box\, \mathit{THROW} \\
\mathit{ReStock} & \stackrel{\text{def}}{=} & \mathit{ReStock}' \,\Box\, \mathit{THROW} \\
\mathit{Transaction} & \stackrel{\text{def}}{=} & [\ \ (\mathit{Order} \div (\mathit{ReStock} \,||\, \mathit{Email}))\,; \\
 & & \quad (\ [\mathit{Offer}_1 \rhd \mathit{Offer}_2] \div \mathit{Withdraw})\,; \\
 & & \quad ((\mathit{Pack} \div \mathit{Unpack}) \,||\, (\mathit{Credit} \div \mathit{Refund}))\,; \\
 & & \quad ((\mathit{Courier}_1 \div \mathit{Cancel}_1) \boxtimes (\mathit{Courier}_2 \div \mathit{Cancel}_2)) \\
 & &] \rhd \mathit{Operator}
\end{array}
$$

Note that cCSP does not offer an explicit construct for alternative forwarding. However, we have used the exception handling operator which achieves the same result if Offer_1 fails. Also, to model the fact that all the involved activities can possibly fail, each activity (eg: *Pack*, *Credit*, *Unpack*, etc) should be defined in a similar fashion to *Order* and *ReStock* where Order' and $\mathit{ReStock}'$ are some lower level activities. Thus, each activity can non-deterministically fail (including compensating activities).

In this example, we have deviated from the original specification in two main aspects: (i) the compensations cannot be executed in parallel because in cCSP the compensations are executed in the reverse order of the original execution; (ii) the interaction among the various entities involved in the transaction (such as the interaction with the client) cannot be modelled.

3.3 StAC

In StAC [CGV+02, BF04], the compensation mechanism is separated from failure. Therefore, both practically and conceptually, compensations need not be used only in case of failure.

Compensations in StAC are stored in so called *compensation stacks* such that compensations can be installed, executed and discarded through stack operations.

StAC_i is an extension of StAC supporting concurrent compensation stacks, implying that several compensation tasks can be maintained concurrently during the execution of a process. Additionally, StAC_i (but not StAC) provides the mechanism for protecting a process from early termination originating from another process. For early termination to propagate (causing forced termination), it must be enclosed within an *attempt* block. Below, we give the StAC_i syntax and a summary of its semantics, further elaborating these notions.

Syntax

Let P range over StAC$_i$ processes; A over atomic activities; b, v over boolean expressions; N over process names; S, X over sets of activities; i over natural numbers; and J over sets of natural numbers.

$$\begin{array}{rcl} P & ::= & A \mid \mathit{skip} \mid b \,\&\, P \mid \mathit{call}(N) \mid P \backslash S \mid P \,;P \mid P \underset{X}{||} P \mid P \sqcap P \mid P \,[\!]\, P \mid \odot \\ & \mid & P\{P\}_v P \mid \mathit{early} \mid \ \mid P \mid_v \mid \mathit{new}(i).P_i \mid \boxtimes_i \mid \boxdot_i \mid \ \uparrow_i P \mid J \rhd i \end{array}$$

A StAC$_i$ process can be an atomic activity A; the inert process *skip*; a guarded process $b \,\&\, P$, which is only activated if the condition, b, is true; a call to another process named N (thus supporting recursion); a process hiding activity P in S (as in CSP [Hoa85]) denoted by $P\backslash S$; a sequential composition of a pair of processes; a parallel composition of two processes, $P \underset{X}{||} P$, synchronising on activities in X; an internal ($P \sqcap P$) or an external choice ($P \,[\!]\, P$) of two processes; early termination $\odot$ (representing failure); an attempt block $P\{P\}_v P$ (explained further on); an early terminated process; a protected block $\mid P \mid_v$ (explained further on); the creation of a new compensation task, i, available in P_i, written as $\mathit{new}(i).P_i$; the execution (in reverse) of a compensation task i, denoted by $\boxtimes_i$; the discarding of a compensation task i, written as $\boxdot_i$; a push of a process P onto a compensation task i, written as $\uparrow_i P$; and the merge of compensation tasks whose index is an element of J onto the compensation task i.

The syntax of StAC$_i$ is quite large and the authors themselves admit that the semantic definition is "somewhat complicated". Having said this, StAC$_i$ offers a high degree of flexibility and expressiveness, particularly, because compensations are separated from failure. This provides the freedom of using compensations as any other programming structure. For example while making a number of related bookings, two kind of compensations are stored: one for confirming the bookings and another for cancelling the temporary bookings. Upon successful completion of all bookings the first set of compensations are triggered, confirming all the bookings, while if a booking fails, the second set of compensations are executing, cancelling all the bookings.

In what follows we attempt to give an overview of the most important aspects of StAC$_i$'s semantics.

Semantics

The first aspect to tackle is the compensation handling mechanism of StAC$_i$. There are five operators related to compensation: $\mathit{new}(i).P_i$, $\boxtimes_i$, $\boxdot_i$, $\uparrow_i P$, and $J \rhd i$. Recall that StAC$_i$ allows multiple compensation tasks to be maintained simultaneously. Each compensation task has a stack structure such that new tasks can be appended (pushed) to one end. A new stack can be created using $\mathit{new}(i).P_i$, meaning that

stack i will be available as a new compensation stack to which tasks can be added throughout P. The operator which pushes process P on stack i is $\uparrow_i P$, signifying that P has been added to the stack. Effectively, if the stack is executed just after the installation of P, P would be the first process to be executed. The execution of compensation tasks in StAC$_i$ has to be programmed explicitly using the operator $\boxtimes_i$, meaning that the operations on stack i are executed in the reverse order of their installation. Similarly, a stack i of compensation tasks can be emptied using $\boxdot_i$ — discarding all the tasks in stack i. Finally, one is allowed to merge a number of compensation tasks with one another through the operator $J \rhd i$. For example, $\{1, 2\} \rhd 3$ means that the parallel composition of compensation tasks 1 and 2 is added sequentially to compensation task 3.

Putting everything together, consider the following example:

$$(A \uparrow_i A')\,;new(j).((B \uparrow_j B')\,;\boxtimes_j\,;(C \uparrow_j C')\,;\boxdot_i\,;\{j\} \rhd i)$$

Upon the first step, A is executed and A' is pushed on compensation task i. Subsequently, a new compensation task j is created and B' is pushed onto j. $\boxtimes_j$ causes the execution of B' and immediately afterwards, C is executed, pushing C' onto j. $\boxdot_i$ causes i to be emptied, discarding compensation A'. Finally, C' is merged onto i, leaving i with C' and discarding j. The executed activities would be $A\,;B\,;B'\,;C$. Note that by having different indexes it is easy to scope compensations, i.e. compensation execution only executes the compensations within the respective compensation task. Also, it is clear that after compensation the process will continue at the point exactly after $\boxtimes$.

Before we proceed to discuss the representation of failure in StAC$_i$, we will explain an inherent exception handling mechanism called the *attempt block* written as $P\{Q\}R$. This operator is similar to a try-and-catch statement where a process Q in curly brackets is tried out and if it succeeds, execution continues with process P on the left-hand side of the block. Otherwise, execution continues with process R on the right. In this way, it combines the ideas behind conditionals and exception handling. For the sake of the semantics, a boolean is attached to the block, flagging whether an exception has been encountered or not. Thus, the full syntax is $P\{Q\}_v R$, with v as the flag. For better readability we use *TRY Q THEN P ELSE R* instead of $P\{Q\}_{false} R$ (The flag is initially false and it is thereafter handled automatically by the semantic rules).

Another construct particular to StAC$_i$ is *early termination*, represented by $\odot$. This is used to signal that a process has ended prematurely. If a process in a sequential composition terminates early, then the rest of the sequential composition is forced to terminate, and the whole sequential composition terminates early. Similarly, if a

process within an attempt block terminates early, then the whole process (possibly including other processes running in parallel) within the attempt block is forced to terminate. The only processes which are not affected by forced termination are the merge ($J \rhd i$) and a protected block (discussed next). However, forced termination does not apply to parallel processes outside attempt blocks. Therefore, attempt blocks can be used to model transactions modelling the behaviour that either the whole transaction succeeds or else it all fails.

A protection block ensures that the block does not get discarded because of a forced termination. The syntax of a protected process P is given by $|\,P\,|_v$ where v indicates that the block has started execution and cannot be interrupted. If the block has not yet started execution, then it is never executed. Protection is notably used to protect compensation operations from interruption. For this reason pushing a compensation onto the stack is usually protected by using the following syntactic sugar: $P \div_i Q \stackrel{\text{def}}{=} |\,P\,;\, \uparrow_i Q\,|_{false}$. Note that upon failure of P, the compensation Q never gets installed.

Another abbreviation used is *IF b THEN P ELSE Q* which is defined as $b \,\&\, P \,[]\, \neg b \,\&\, Q$. Note that external choice ($[]$) chooses whichever of the processes becomes ready first. In cases where both processes can execute simultaneously (eg. $A\,;B$ and $A\,;C$), internal choice ($\sqcap$) should be used.

In StAC$_i$, compensations can have compensations. For example $A \div (B \div C)$ is a valid process, installing compensation $B \div C$ upon the execution of A and installing C upon the execution of the compensation task B.

Example

The example of the bookstore encoded in StAC$_i$ is given in parts. We start with the definition of the basic actions, using primed names to represent lower level activities with the actual logic:

$$\begin{aligned}
\mathit{Order} &\stackrel{\text{def}}{=} \mathit{Order}' \,[]\, (\boxtimes_0 \,;\odot)\\
\mathit{Pack} &\stackrel{\text{def}}{=} \mathit{Pack}' \,[]\, (\boxtimes_0 \,;\odot)\\
\mathit{Offer}_1 &\stackrel{\text{def}}{=} \mathit{Offer}_1{}' \,[]\, (\boxtimes_1 \,;\odot)\\
\mathit{Offer}_2 &\stackrel{\text{def}}{=} \mathit{Offer}_2{}' \,[]\, (\boxtimes_1 \,;\odot)\\
\mathit{ReStock} &\stackrel{\text{def}}{=} \mathit{ReStock}' \,[]\, \odot
\end{aligned}$$

Recall that the example requires that any of the activities might fail. To model such behaviour we need to provide an external non-deterministic choice, which in case of failure, compensates the previous activities (using $\boxtimes_i$) and terminates execution (using $\odot$). This is the case of *Order*, *Pack*, etc. Note that the compensation context

i depends on the context of the execution of the process. For example, in the case of *Order* context 0 is used, while in the case of $Offer_1$ context 1 is used. If compensating activities fail, we opt to simply signal a failure ($\odot$). Other compensating activities should be defined in a similar fashion to *ReStock*.

For the case of the offers we have used two nested *TRY* statements such that if one fails, the other is tried out.

$$\begin{array}{lll} Offers & \stackrel{\text{def}}{=} & new(1).(\\ & & \quad TRY\ Offer_1 \div_0 Withdraw \\ & & \quad THEN\ skip \\ & & \quad ELSE\ (TRY\ Offer_2 \div_0 Withdraw\ THEN\ skip\ ELSE\ skip)) \end{array}$$

The new stack created via $new(1)$ ensures that if any of the offers fail, it is not the outer compensation that is executed (since $\boxtimes_1$ is used within $Offer_1$ and $Offer_2$ instead of $\boxtimes_0$ as explained above). Note that if $Offer_2$ fails, no failure is signalled since the failure is caught by the *TRY* statement and continues as *skip*.

The most complex part of the example is the speculative choice of the couriers as shown below:

$$\begin{array}{lll} Couriers & \stackrel{\text{def}}{=} & TRY\ new(2).new(3).(\\ & & \quad \mid Courier_1\ ;\ \uparrow_2 Cancel_1 \mid_{true} \mid\mid \mid Courier_2\ ;\ \uparrow_3 Cancel_2 \mid_{true} \\ & & \quad \mid\mid (Ready_1\ \&\ \odot) \mid\mid (Ready_2\ \&\ \odot) \\ & &)\ THEN\ skip \\ & & ELSE\ (IF\ Ready_1\ THEN\ \boxtimes_3\,;\{2\} \rhd 0 \\ & & \quad ELSE\ (IF\ Ready_2\ THEN\ \boxtimes_2\,;\{3\} \rhd 0\ ELSE\ \boxtimes_0\,;\odot)) \end{array}$$

In order to encode speculative choice, we required two extra boolean variables: $Ready_1$ and $Ready_2$. These become true when $Courier_1$ or $Courier_2$ succeed, respectively. Note that the booking of the couriers is put inside a *TRY* block and in parallel to the booking processes, we added two processes which signal early termination upon the completion of any of the bookings. Thus, when a booking succeeds, the whole process is terminated early. Furthermore, upon early termination, if both bookings succeeded simultaneously, one has to be compensated while the compensation of the other has to be relayed to the outer compensation (using $\rhd 0$). Note that if one of the bookings terminates early (due to failure), the other booking is not affected since both are enclosed in a protected block with the boolean flag set. If neither booking succeeds, the compensation is executed and early termination is signalled so that the outer *TRY* statement executes the *Operator* process.

Finally, the overall transaction process is given below:

$$
\begin{aligned}
\mathit{Transaction} \stackrel{\text{def}}{=}\ & \mathit{TRY}\ (\ (\mathit{Order} \div_0 (\mathit{ReStock} \,||\, \mathit{Email}))\,; \\
& \quad \mathit{Offers}\,; \\
& \quad ((\mathit{Pack} \div_0 \mathit{Unpack}) \,||\, (\mathit{Credit} \div_0 \mathit{Refund}))\,; \\
& \quad (\mathit{Couriers}) \\
&)\ \mathit{THEN}\ \mathit{skip} \\
& \mathit{ELSE}\ \mathit{Operator}
\end{aligned}
$$

Note that the transaction is enclosed within a *TRY* block so that if it results in an early termination (due to a failure), the operator is notified. Note that since there is no way of distinguishing between failure during normal execution and failure during compensation, the operator is notified even when compensation is successful, diverging from the specification of the example.

As in the case of Sagas and cCSP, due to the stack structure which StAC$_i$ uses for storing compensations, there seems to be no straightforward way of running the compensations in parallel. Also, it is not possible to model the interactions among the parties involved in the transaction.

3.4 Transaction Calculus

The transaction calculus (t-calculus) [LZH07, LZPH07] is a highly expressive language which builds on the ideas of StAC [CGV$^+$02, BF04], cCSP [BHF04, BR05], and Sagas [BMM05], providing an algebraic semantics for transactions. The transaction calculus provides various exception handling mechanisms apart from the compensation mechanism. The building blocks of the calculus are atomic actions which always succeed. Each of these actions has a compensation action associated to it which can be the empty process or the process which always fails. Basic actions with compensations are connected together to form transactions, which in turn can be enclosed as a transaction block.

Syntax

We will use A and B to range over a set of basic activities Σ, variables S and T to range over transactions, and variable P to range over transaction blocks:

$$
\begin{array}{rcll}
BT & ::= & A \div B \mid A \div 0 \mid A \div \Diamond & \text{basic transactions} \\
 & \mid & \mathit{Skip} \mid \mathit{Abort} \mid \mathit{Fail} & \\
S, T & ::= & BT & \text{basic transaction} \\
 & \mid & S \,;\, T & \text{sequential composition} \\
 & \mid & S \mid\mid T & \text{parallel composition} \\
 & \mid & S \,[\!]\, T & \text{external choice} \\
 & \mid & S \sqcap T & \text{internal choice} \\
 & \mid & S \otimes T & \text{speculative choice} \\
 & \mid & S \trianglerighteq T & \text{backward exception handling} \\
 & \mid & S \rhd T & \text{forward exception handling} \\
 & \mid & S \rightsquigarrow T & \text{alternative forwarding} \\
 & \mid & S \circledast T & \text{programmable compensation} \\
P & ::= & \{T\} & \text{transaction block}
\end{array}
$$

A basic transaction may be a basic activity A with possibly another action B as compensation, written as $A \div B$; a basic activity A with an empty compensation which always succeeds (0); a basic activity A with an empty compensation which always fails ($\Diamond$); a *Skip* signifying a successful transaction; an *Abort* signifying a transaction which needs to be compensated; or a *Fail* signifying a failed transaction which cannot be compensated and is treated as an exception.

A transaction may either be a basic transaction BT; a parallel or a sequential composition; an external, internal or speculative choice (denoted by $S \,[\!]\, T$, $S \sqcap T$, and $S \otimes T$ respectively); a transaction S with a backward exception handler T, written as $S \trianglerighteq T$, meaning that after the execution of the exception handler, the transaction should be considered as never executed; a transaction S with a forward exception handler T, written as $S \rhd T$ such that after the execution of the exception handler, the transaction should be considered accomplished; an alternative forwarding denoted by $S \rightsquigarrow T$ such that if S compensates, T is tried out as an alternative; or a transaction with programmable compensation denoted by $S \circledast T$ such that T replaces the compensation of S. Finally, a transaction block represented a transaction enclosed within curly brackets.

In t-calculus there are two operators for attaching compensations: $\div$ is used to attach a compensation to a basic activity, while $\circledast$ attaches a transaction as a compensation of another transaction (replacing the existing compensation). For example, we can write: $A \div A' \circledast B \div B'$. However, note that the compensation of the compensation (i.e. B') is semantically discarded.

Semantics

In order to give the formal definition, we introduce a number of preliminaries which

will be used further on.

A one-step behaviour is denoted by $h \rightarrow T$ where the activity h can either be: (i) an atomic activity; (ii) an error activity — $\Diamond$ if the error occurs during the forward flow, $\overline{\Diamond}$ if otherwise; (iii) a ready activity $\sharp$; or (iv) a forced activity $\lnot$ which leads to forced termination.

There are five states in which a transaction can terminate: (i) $\boxdot\backslash T$ signifies that a transaction has successfully terminated having T as its compensation; (ii) $\boxtimes$ denotes a successfully compensated transaction; (iii) $\overline{\boxtimes}$ denotes a forced abort which has been successful; (iv) $\circledast$ represents a failed transaction whose compensation also failed; and (v) $\overline{\circledast}$ represents a transaction whose forced compensation failed.

In the semantic rules which follow the following operators are used: (i) $S \uparrow T$ means that S is next to be executed in the forward flow with T as the currently installed compensation; (ii) $\overleftarrow{T}$ represents a compensation T which has been automatically activated upon failure; (iii)$\overleftrightarrow{T}$ represents a compensation T which has been forcedly activated.

In order to give the algebraic laws, transactions are converted into their *head normal form* (hnf) using the function $\mathcal{HF}()$. A transaction is in hnf if either: (i) it is in $\{\boxdot\backslash T, \boxtimes, \overline{\boxtimes}, \circledast, \overline{\circledast}\}$; (ii) it is of the form $[]_{i \leq N} h_i \rightarrow T_i$ such that each T_i is in hnf; or (iii) it is of the form $\sqcap_{j \leq M} T_j$ such that each T_j is in hnf.

In the case of the simple transaction $A \div B$, either it is forced to terminate evolving to $\overline{\boxtimes}$, requiring no compensation, or else A succeeds and compensation B is installed:

$$\mathcal{HF}(A \div B) \;=\; \lnot \rightarrow \overline{\boxtimes} \;[]\; A \rightarrow \mathcal{HF}(Skip \uparrow (B \div 0))$$

The case of $\mathcal{HF}(A \div 0)$ is similar but there is no compensation to be installed. However, in the case of $\mathcal{HF}(A \div \Diamond)$, the compensation installed is *Abort* because it always fails during the backward flow. For a *Skip*, *Abort* or *Fail* the rules are given below such that a transaction succeeds ($\boxdot$), aborts ($\boxtimes$) or fails ($\circledast$) respectively unless there is a forced termination:

$$\begin{array}{rcl}
\mathcal{HF}(Skip) & = & \lnot \rightarrow \overline{\boxtimes} \;[]\; \sharp \rightarrow \boxdot\backslash Skip \\
\mathcal{HF}(Abort) & = & \lnot \rightarrow \overline{\boxtimes} \;[]\; \Diamond \rightarrow \boxtimes \\
\mathcal{HF}(Fail) & = & \lnot \rightarrow \overline{\boxtimes} \;[]\; \Diamond \rightarrow \circledast
\end{array}$$

Next, we consider what happens after the termination of a transaction, considering the possible termination modes:

$$\frac{\mathcal{HF}(S) = \boxdot \backslash S'}{\mathcal{HF}(S \uparrow T) = \boxdot \backslash (S'; T)} \qquad \frac{\mathcal{HF}(S) = S', S' \in \{\circledast, \overline{\circledast}\}}{\mathcal{HF}(S \uparrow T) = S'}$$

The first rule states that upon the successful completion of a transaction S, the compensation is sequentially installed with the accumulated compensation T. Otherwise (by the second rule), if a transaction fails, the parent transaction fails as well. Next, we consider the other two cases of transaction termination:

$$\frac{\mathcal{HF}(S) = \boxtimes}{\mathcal{HF}(S \uparrow T) = \mathcal{HF}(\overleftarrow{T})} \qquad \frac{\mathcal{HF}(S) = \overline{\boxtimes}}{\mathcal{HF}(S \uparrow T) = \mathcal{HF}(\overleftrightarrow{T})}$$

If a transaction aborts, then the outcome is the same as that of the installed compensation. Similarly, in the case of a forced abort, the outcome is the same as that of the installed compensation. The following two rules cater for transactions which are executed as compensations:

$$\frac{\mathcal{HF}(T) = \boxdot \backslash T'}{\mathcal{HF}(\overleftarrow{T}) = \boxtimes} \qquad \frac{\mathcal{HF}(T) = T', T' \in \{\boxtimes, \circledast\}}{\mathcal{HF}(\overleftarrow{T}) = \circledast}$$

A successful compensation implies that the transaction has been correctly compensated (i.e. aborted ($\boxtimes$)). Otherwise, if the compensation aborts or fails, then the transaction fails ($\circledast$). (The case of a forced termination is similar but the rules are not given for brevity.)

From the above rules, the similarity of t-calculus to Sagas and cCSP is immediately apparent. The same applies for the handling of parallel transactions. If two parallel transactions succeed, then their compensations are installed in parallel. When both fail, the end result depends on a special function (&) which follows the following two simple rules: if one of the parallel transaction fails, then the whole parallel composition fails and if one of the parallel transaction is not a forced termination, then the parallel composition in not a forced termination.

Of particular interest is the definition of the programmable compensation, speculative choice, exception handling mechanisms and alternative forwarding because these are rarely explicitly available in other calculi. In the case of programmable compensation, upon the successful completion of a transaction, the existing accumulated compensation is discarded and the programmable compensation is installed. Otherwise, the accumulated compensation is not discarded.

Alternative forwarding is interesting because it provides a continuation point after successful abortion. Without such an operator (as in the case of many other calculi) there is no way (except through scoping[4]) of going back to the forward flow once the backward flow has been started. Note that if T is the alternative of S, T is only run if S aborts. Otherwise, if S succeeds (triggering no compensation) or fails during compensation, T is never activated.

t-calculus offers two ways of handling exceptions: forward or backward. The forward exception handler is activated only upon a non-forced failure, i.e. when a compensation, which has not been triggered by forced termination, fails. Otherwise, the exception handler is not activated. Note that forward exception handling is similar to alternative forwarding which instead of activating upon failure, activates upon abort. In the case of backward exception handling, the exception handler is activated as backward behaviour upon a failure (forced or not), i.e. the transaction continues with compensation after the termination of a backward exception handler.The rules which handle forward activation, backward activation in the case of forced and non-forced termination are given below:

$$\frac{\mathcal{HF}(S) = \circledast}{\mathcal{HF}(S \rhd T) = \mathcal{HF}(T)} \qquad \frac{\mathcal{HF}(S) = \overline{\circledast}}{\mathcal{HF}(S \trianglerighteq T) = \mathcal{HF}(\overleftrightarrow{T})} \qquad \frac{\mathcal{HF}(S) = \circledast}{\mathcal{HF}(S \trianglerighteq T) = \mathcal{HF}(\overleftarrow{T})}$$

The first rule deals with forward exception handling in the case of a failure. The second and third rules deal with backward exception handling of forced and non-forced failure respectively. Note that in the latter cases, transaction T is executed as backward behaviour.

Example

The example of the bookstore given in t-calculus is as follows:

$$\begin{aligned}
\mathit{Order} &\stackrel{\text{def}}{=} (\mathit{Order}' \div 0) \sqcap \mathit{Abort} \\
\mathit{ReStock} &\stackrel{\text{def}}{=} (\mathit{ReStock}' \div \Diamond) \sqcap \mathit{Fail} \\
\mathit{Transaction} &\stackrel{\text{def}}{=} \{\ (\ (\mathit{Order} \circledast (\mathit{ReStock} \,||\, \mathit{Email}))\,; \\
&\qquad ((\mathit{Offer}_1 \rightsquigarrow \mathit{Offer}_2) \circledast \mathit{Withdraw})\,; \\
&\qquad ((\mathit{Pack} \circledast \mathit{Unpack}) \,||\, (\mathit{Credit} \circledast \mathit{Refund}))\,; \\
&\qquad ((\mathit{Courier}_1 \circledast \mathit{Cancel}_1) \otimes (\mathit{Courier}_2 \circledast \mathit{Cancel}_2)) \\
&\qquad) \rhd \mathit{Operator}\ \}
\end{aligned}$$

[4]Recall that abortion is not propagated to the parent in the case of Sagas and cCSP.

Note that the undefined processes ($Offer_1$, $Pack$, etc) can be defined similarly to $Order$ where we encode the possibility of failure by a non-deterministic internal choice between the execution of the activity and $Abort$. With this arrangement we model the possibility of an action failing since in the semantics of the t-calculus, basic activities (such as $Order'$) never fail. In the case of compensating activities such as $ReStock$, $Email$, etc, the definition is similar to the one given for $ReStock$. Note that if the compensation fails, this time we signal a $Fail$ rather than an abort, meaning that the problem cannot be compensated (unlike $Abort$).

The definition of the main transaction is quite straightforward due to the rich syntax of the t-calculus, providing explicit operators for alternative forwarding, speculative choice and programmable compensation.

Note that in this example we have also failed to model the interactions among the transaction parties and the compensations cannot run in parallel as requested by the specification.

3.5 Automata

Automata have also been used to model compensating transactions. Apart from the advantage of being graphical, a lot of work has already been done in automata particularly in the area of verification which can then easily be adapted for compensations.

Communicating hierarchical transaction-based timed automata (CHTTAs) [LMSMT06, LMSMT08] are communicating hierarchical machines [AKY99] enriched with time (similarly to timed automata [AD94]), and with other slight modifications to accommodate the representation of transactions. Two appealing features of CHTTAs (apart from the inherent graphical aspect) is that they support the notion of time and can be reduced to timed automata and hence model-checkable. Long running transactions (LRTs) are defined over and above CHTTAs such that a CHTTA can be specified as the compensations of another CHTTA. Furthermore, LRTs can also be nested or composed in parallel or sequentially. Similarly to a number of other approaches, the order of compensation execution in LRTs is in reverse order in case of sequence and in parallel in case of a parallel composition. Also, in the case of successfully aborted nested transactions, the parent transaction is not aware of abortion and continues unaffected. A limitation of LRTs is that they do not show clearly (graphically) which compensation corresponds to which component and it is assumed that compensations succeed. The latter limitation can be lifted by introducing exception handling which is completely absent in LRTs. Another mechanism which LRTs do not provide is forced termination and consequently neither termination handling.

In what follows, we give a brief overview of the syntax and semantics of LRTs: (i)

first, we introduce the basic transaction-based timed automata (TTAs) on top of which CHTTAs are defined; (ii) next, we introduce CHTTAs; and (iii) finally, we describe how CHTTAs can be composed together through nesting, sequential and parallel composition to form LRTs.

Transaction-based Timed Automata

A transaction-based timed automaton (TTA) is a timed automaton with some additions enabling it to handle transactions. Formally, given a set of communicating channels $\mathcal{C}$, a set of clocks X, and a set of clock constraints $\Phi(X)$ on X, a TTA is a tuple $A = (\Sigma, X, S, Q, q_0, Inv, \delta)$ where:

$\Sigma \subseteq \{a!, a? \mid a \in \mathcal{C}\}$ is a finite set of labels;
S is a finite set of superstates;
L is a finite set of basic states;
$Q = L \cup S \cup \{\odot, \otimes\}$ where $\odot$ and $\otimes$ represent the special states commit and abort respectively;

$q_0 \in L$ is the initial state;
$Inv : L \cup S \to \Phi(X)$ assigns a function to each state which must hold whenever the state is enabled;
$\delta \subseteq (L \times \Sigma \cup \{\tau\} \times \Phi(X) \times 2^X \times Q) \cup (S \times \{\boxdot, \boxtimes\} \times Q)$ is the set of transitions with special labels $\boxdot$ and $\boxtimes$ being special labels for commit and abort transitions respectively.

Note the various modifications to timed automata: (i) there are two different kinds of states: superstates and basic states; (ii) the introduction of two special states, $\odot$ and $\otimes$, which are considered as final states; and (iii) special transitions labelled with $\boxdot$ and $\boxtimes$ which are the only transitions possible from superstates. The motivation behind these additions we be clear when we introduce CHTTAs and LRTs in what follows.

Communicating Hierarchical Transaction-based Timed Automata

Given a set of TTAs $\mathcal{A} = \{A^1, \ldots, A^n\}$ (with $A^i = (\Sigma^i, X^i, S^i, Q^i, q_0^i, Inv^i, \delta^i)$), a CHTTA is either a tuple $\langle A^i, \mu \rangle$, or a parallel composition of two CHTTAs, where μ is a hierarchical composition function $u : S^i \to CHTTA_{\{A^{i+1}, \ldots, A^n\}}$, assigning a CHTTA to each superstate. Note that a state can only be refined by an automaton with a higher index, avoiding cyclic nesting. Thus, CHTTAs are parallel compositions of TTAs with superstates refined in terms of other TTAs. Through shared channels, parallel CHTTAs can communicate by synchronising transitions with a sending and a receiving action on a particular channel.

Although CHTTAs support special states and transition labels to denote transaction commit and success, to support compensations another layer on top of CHTTAs is required; enabling the definition of long running transactions.

Long Running Transactions

A long running transaction (LRT) requires that upon the failure of one of its activities, any previously completed activities are compensated. Thus, the basic building block of a LRT is the association of a CHTTA as the compensation of another: a CHTTA, B, can be specified as the compensation of another CHTTA, A, represented by $A \Rsh B$. Furthermore, LRTs can be composed: (i) sequentially — having two LRTs, T_1 and T_2, $T_1 \cdot T_2$ is their sequential composition; (ii) in parallel — having two LRTs, T_1 and T_2, $T_1 \mid\mid T_2$ is their parallel composition. For example, if A_2 in $A_1 \Rsh B_1 \cdot A_2 \Rsh B_2$ fails, then B_1 must be executed to compensate for A_1. However, through nesting, a transaction may have subtransactions such that if a subtransaction fails but successfully compensates, then the main transaction is not aborted. A nested transaction is represented by $\{T\}$ where T is a LRT.

Semantics

The semantics of LRTs is given in terms of an encoding function $[\![\cdot]\!] : T \rightarrow CHTTA \times CHTTA \times \mathcal{P}(CHTTA)$, which given a LRT returns (i) a CHTTA representing the forward behaviour; (ii) a CHTTA representing the backward (compensating) behaviour; (iii) a set of CHTTAs, one for each nested transaction, controlling whether the compensation for a nested transaction needs to be executed (compensation is only executed if the nested transaction has not aborted). It is beyond the scope of this overview to give the full semantics of LRTs but to give an idea, we show how the semantics of two sequentially composed long running transactions is defined.

Informally, the LRT $T = A_1 \Rsh B_1 \cdot \{A_2 \Rsh B_2\} \cdot A_3 \Rsh B_3$ (where $A_1, A_2, A_3, B_1, B_2, B_3$ are CHTTAs) should (i) execute A_1 followed by A_2 followed by A_3; (ii) but if A_1 fails, execution stops; (iii) while if A_2 fails, execution progresses normally to A_3 since A_2 is nested; (iv) if A_3 fails, A_2 has to be compensated (if it has not failed) and A_1 also has to be compensated; (v) if all of A_1, A_2 and A_3 succeed, then T commits and it can be undone through the compensation B_3 followed by B_2 followed by B_1. Thus $(A, B, M) = [\![T]\!]$ is formally defined as (depicted in Figure 2):

$$A = \langle(\emptyset, \emptyset, \{s_1, \ldots, s_5\}, \{s_1, \ldots, s_5, q_0, \odot, \otimes\}, q_0, Inv_{true}, \delta), \mu\rangle$$
$$B = \langle(\emptyset, \emptyset, \{s_1, s_2, s_3\}, \{s_1, s_2, s_3, q_0, \odot, \otimes\}, q_0, Inv_{true}, \delta'), \mu'\rangle$$
$$M = \emptyset$$

where

$\delta = \{(q_0, \tau, true, \emptyset, s_1), (s_1, \boxdot, s_2), (s_1, \boxtimes, \otimes), (s_3, \boxdot, \odot), (s_3, \boxtimes, s_4), (s_2, \boxdot, s_3), (s_4, \boxdot, s_5), (s_5, \boxdot, \otimes)\}$
$\mu = \{(s_1, A_1), (s_2, A_2'), (s_3, A_3), (s_4, B_2'), (s_5, B_1)\}$
$\delta' = \{(q_0, \tau, true, \emptyset, s_3), (s_3, \boxdot, s_2), (s_2, \boxdot, s_1), (s_1, \boxdot, \odot)\}$
$\mu' = \{(s_1, B_1), (s_2, B_2), (s_3, B_3), (s_4, B_2')\}$
$A_2' = \langle(\{cn_1!, an_1!\}, \emptyset, \{s_1\}, \{s_1, q_0, q_1, q_2, \boxdot\}, Inv_{true}, \delta_A), \mu_A\rangle$
$B_2' = \langle(\{cn_1?, an_1?\}, \emptyset, \{s_1\}, \{s_1, q_0, \boxdot\}, Inv_{true}, \delta_B), \mu_B\rangle$
$\delta_A = \{(q_0, \tau, true, \emptyset, s_1), (s_1, \boxdot, q_1), (s_1, \boxtimes, q_2), (q_1, cn_1!, true, \emptyset, \odot), (q_2, an_1!, true, \emptyset, \odot)\}$
$\mu_A = \{(s_1, A_2)\}$
$\delta_B = \{(q_0, cn_1?, true, \emptyset, s_1), (s_1, \boxdot, q_1), (q_0, an_1?, true, \emptyset, \odot)\}$
$\mu_B = \{(s_1, B_2)\}$

Next, we encode the bookshop example in terms of a LRT.

Example

There are a number of limitations in long running transactions which do not allow the faithful specification of the example. Unless one adds more operators through channel communication, features such as exception handling, alternative forwarding and speculative choice are not available. Thus the bookshop example in long running transactions can be encoded a follows:

$$(Order \triangleright (ReStock \mid\mid Email)) \cdot \{Offer_1 \triangleright Withdraw\} \cdot$$
$$((Pack \triangleright Unpack) \mid\mid (Credit \triangleright Refund)) \cdot (Courier_1 \triangleright Cancel_1)$$

Note that the following features could not be encoded: (i) alternative forwarding to try the second offer if the first one fails; (ii) speculative choice among the couriers; and (iii) exception handling reporting failure to a human operator.

The basic activities such as *Order* and *ReStock* can be encoded as a CHTTA having channel communication with a third-party which then communicates back on other channels indicating whether the action was successful or not. This is depicted in Figure 3.

Note that there are no superstates and thus the refinement function μ is empty.

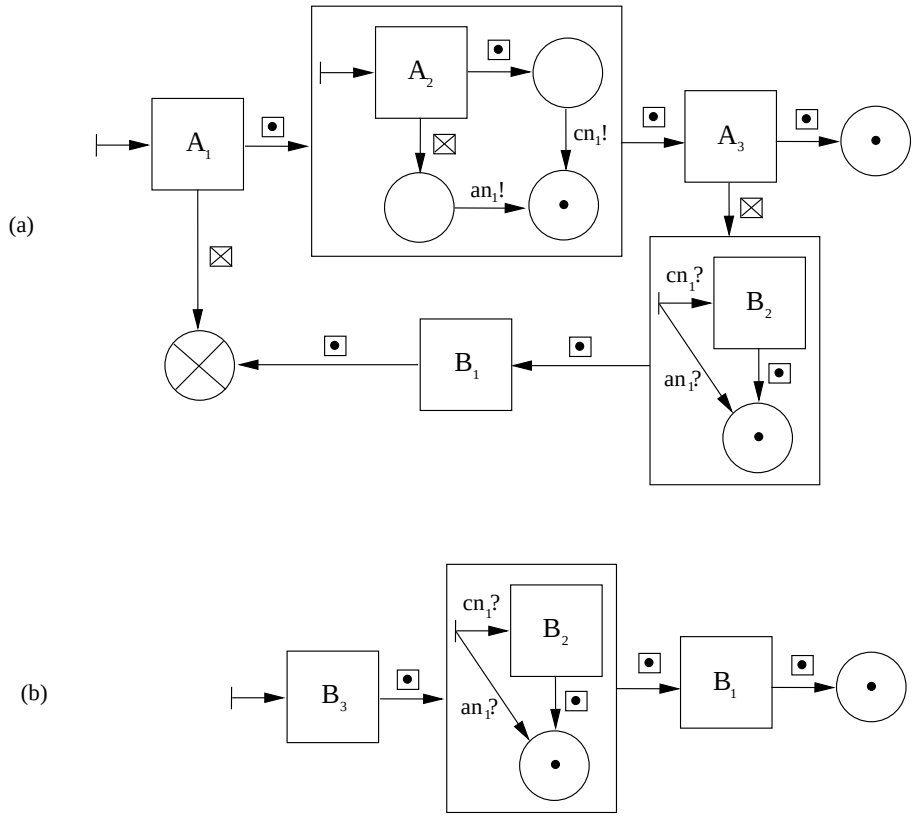

Figure 2: A representation of $(A, B, M) = [\![T = A_1 \Gamma B_1 \cdot \{A_2 \Gamma B_2\} \cdot A_3 \Gamma B_3]\!]$ with (a) showing A and (b) showing B.

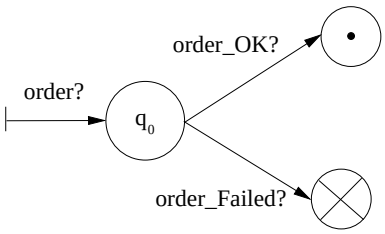

Figure 3: A representation of the *Order* CHTTA.

3.6 Comparison of Orchestration Approaches

The majority of orchestration approaches are quite similar, particularly Sagas, cCSP, and t-calculus basically vary as regards some operators which one offers and the others do not (ignoring more technical aspects such as the way the semantics are

defined). For example cCSP offers the yield operator while t-calculus offers the backward exception handler. An orchestration approach which particularly stands out is StAC due to its treatment of compensations as a series of stack operations. Thus, operations which in the other notations are automatic (e.g. compensation installation and activation), in StAC are explicitly programmed by the programmer. The automata approach stands out mostly due to the fact that it is automata-based and not due to major differences in the underlying semantics. Apart from this, it differs because it does not support exception handling and forced termination. Finally, BPEL also shares many commonalities with the other orchestration approaches. The main difference is that BPEL allows for full customisation of the compensation and that compensations are executed in reverse order of the original execution (as opposed to running compensations in parallel for a parallel forward execution).

Other features which are common to all notations will be discussed in Section 6 where orchestration approaches are compared to choreography approaches.

4 Choreography Approaches

As opposed to orchestration approaches, choreography approaches do not assume a centralised coordination mechanism. Rather, activities participating in a choreography collaborate together towards forming a higher-level activity through direct communication. This means that each activity is aware of other collaborators and knows when to start processing (according to some communication sequence). Due to the importance of communication in choreography approaches it is not surprising to find that all the approaches reviewed in this section are process algebras. This is mainly because communicating processes can be naturally modelled in process algebras.

In what follows, we present a number of process algebras which model a distinctive compensation modelling approach. Other process algebras which are intended as formalisations of BPEL will be tackled in the section dedicated to BPEL (Section 5.3.3).

4.1 πt Calculus

$\pi\mathbf{t}$ calculus [BLZ03] can be considered as one of the first attempts of formalising long running transactions using a process algebras. The approach is quite different from that of flow composition languages — mainly due to the distributed approach of processes rather than a central approach.

$\pi\mathbf{t}$ calculus shares many constructs with the π calculus with the addition of the transaction construct which is not present in the π calculus. Basically, a transaction (in $\pi\mathbf{t}$ calculus) is a process with another three associated processes: a failure manager, a compensation store, and a compensation process. The failure manager is a process which is executed in case the transaction fails, the compensation store is a collection of compensations which compensate for previously completed nested transactions, while the compensation process is a process which will be added to the compensation store if the current transaction succeeds. Then, in case of a failure, the compensation store is executed followed by the execution of the failure manager.

Syntax

The following is the syntax of the $\pi\mathbf{t}$ calculus with P ranging over processes, x, y, u, v ranging over a countable set of names, $\widetilde{u}$ ranging over tuples of names, K, K' ranging over process constants, a ranging over boolean variables, and k over boolean values:

$$
\begin{array}{rcll}
P & ::= & \mathit{done} & \text{success} \\
 & | & \mathit{abort} & \text{error} \\
 & | & \overline{x}\langle\widetilde{u}\rangle & \text{output} \\
 & | & x(\widetilde{u}).P & \text{input} \\
 & | & P \mid P & \text{parallel} \\
 & | & P\,;P & \text{sequence} \\
 & | & (x)P & \text{new} \\
 & | & K(\widetilde{u}) & \text{invocation} \\
 & | & \mathit{if}\ (a = k)\ \mathit{then}\ P\ \mathit{else}\ P & \text{conditional} \\
 & | & \mathbf{t}(P, P, P, P) & \text{transaction}
\end{array}
$$

Thus, a $\pi\mathbf{t}$ calculus process can be *done* or *abort* being the inert process or the failing process respectively; $\overline{x}\langle\widetilde{u}\rangle$, representing an output on channel x affecting a set of variables $\widetilde{u}$; $x(\widetilde{u}).P$, denoting input on channel x affecting the set of variables $\widetilde{u}$; $P|P$ and $P\,;P$ representing parallel and sequential composition respectively; $(x)P$, specifying name x as local for the given scope P; $K(\widetilde{u})$, being the invocation of a process P with values $\widetilde{u}$ where $K(\widetilde{u}) = P$; the conditional operator, which executes the first process if $a = k$ or the second if otherwise, or the transaction process $\mathbf{t}(P, F, B, C)$ where P is the main process, F is the failure handler, B is the accumulation of compensations, and C is the compensation process for the given transaction.

Semantics

The semantics of $\pi\mathbf{t}$ calculus is given in terms of reduction rules. In this short overview, we focus mostly on those concerning compensation. The first one which provides a lot of insight as to the handling of transactions is the following:

$$\mathbf{t}(\mathbf{t}(\mathit{done}, F, B, C) \mid P, F', B', C') \rightarrow \mathbf{t}(P, F', B' \mid C, C')$$

The intuition behind it is quite straightforward. Upon the successful completion of a transaction (terminating as *done*), the corresponding compensation is added in parallel to the "compensation bag" and control is passed on to the parent transaction. The fact that compensations are added in parallel implies that the only way of having sequentiality among compensations is through the use of synchronisation. This approach will similarly be utilised in other process algebras. Also note that the accumulated compensation B is discarded and not propagated to the parent transaction. This means that the accumulated compensation of the completed transaction is being discarded and replaced by C.

After we have considered what happens upon the successful termination of a transaction, next we consider the possibility of failure by considering the following rule:

$$\mathbf{t}(abort, F, B, C) \rightarrow B\,;F$$

This rule shows that upon the failure of a transaction, both the accumulated compensations and the failure manager are executed in sequence ($B\,;F$). Hence upon an abort within a transaction, the parent (if there is one) is not immediately aware of the failure, but will be if $B\,;F$ also fails.

Next, we consider how a parent transaction is affected by the failure of one of its children. For this purpose, we explain the following two structural congruence equations:

$$\begin{array}{rcl} abort \mid abort & \equiv & abort \\ done \mid P & \equiv & P \end{array}$$

These rules (and the absence of the rule $abort \mid P \equiv abort$) implies that for a parallel composition to be treated as a failed process, all the processes must terminate (i.e. there is no notion of forced termination). Only then, can the rule which handles the transaction abort trigger and cause the compensation bag to be executed. Note that one aborted process is enough for the whole transaction to be considered aborted since by the second equation $done \mid abort \equiv abort$.

Example

The example in $\pi\mathbf{t}$-calculus is given in parts; we start by showing how basic actions can be encoded:

$$\begin{array}{rcl} Store & \stackrel{\text{def}}{=} & order(resp).\overline{resp}(1) \mid order(resp).\overline{resp}(0) \\ Store' & \stackrel{\text{def}}{=} & stock(resp).\overline{resp}(1) \mid restock(resp).\overline{resp}(0) \\ Order & \stackrel{\text{def}}{=} & \overline{order}\langle resp\rangle \mid resp(a).if\ (a=1)\ then\ done\ else\ abort \mid Store \\ ReStock & \stackrel{\text{def}}{=} & \overline{restock}\langle resp\rangle \mid resp(a).if\ (a=1)\ then\ done\ else\ abort \mid Store' \\ Shop & \stackrel{\text{def}}{=} & \mathbf{t}(Order, done, done, (ReStock \mid Email)) \end{array}$$

Note how the *Shop* transaction is responsible for starting the stock handling activity — we exploit the transaction construct with the first element being the actual activity (*Order*) and the last entry of the quadruple being the corresponding compensation (*ReStock* in parallel with *Email*). Furthermore, the *Order* operation is encoded as a communication with the *Store* (representing an activity at the book warehouse) such that if the activity at the store succeeds, the *Order* activity receives a positive response. The possibility of failure is encoded by non-deterministically sending both a 1 and a 0. The same approach should be used for encoding the rest of the activities.

Next we show how the offers are tackled:

$$\begin{array}{lcl} \mathit{Offers}_1 & \stackrel{\text{def}}{=} & \mathbf{t}(\mathit{Offer}_1, \mathit{Offers}_2, \mathit{done}, \mathit{Withdraw}) \\ \mathit{Offers}_2 & \stackrel{\text{def}}{=} & \mathbf{t}(\mathit{Offer}_2, \mathit{done}, \mathit{done}, \mathit{Withdraw}) \end{array}$$

In this case, we use the failure manager of the transaction which is specified as the second entry in the transaction definition. In this way we model the behaviour that if Offer_1 fails, Offer_2 is started, and if Offer_2 fails as well, execution continues since the failure handler always succeeds. Note that conceptually the turning down of an offer is not a failure however the encoding presented above provides the equivalent behaviour in the absence of an alternative forwarding operator.

Finally, we tackle the courier booking:

$$\begin{array}{lcl} \mathit{Couriers} & \stackrel{\text{def}}{=} & \mathbf{t}(\overline{\mathit{Courier}_1}\langle \mathit{resp}_1 \rangle \mid \overline{\mathit{Courier}_2}\langle \mathit{resp}_2 \rangle \mid \\ & & \quad \mathit{resp}_1(a_1).\mathit{resp}_2(a_2). \\ & & \quad\quad \mathit{if}\ (a_1 = 1\ \&\ a_2 = 1) \\ & & \quad\quad \mathit{then}\ \mathit{Cancel}_2\,;\mathit{done} \\ & & \quad\quad \mathit{else\ if}\ (a_1 = 0\ \&\ a_2 = 0) \\ & & \quad\quad\quad\quad \mathit{then\ abort\ else\ done}, \\ & & \quad \mathit{done}, \mathit{done}, \mathit{Cancel}) \end{array}$$

This is the most intricate part of the example. The values received on channels resp_1 and resp_2 model the response received from $\mathit{Courier}_1$ and $\mathit{Courier}_2$ respectively. Subsequently, if both bookings succeed, then one is cancelled (by executing Cancel_i). If both fail, then the courier booking fails. Otherwise (if one succeeds and one fails), nothing is done except reporting success. Note that it is not straightforward to encode the decision of whether to use Cancel_1 or Cancel_2 as a compensation for the whole transaction. Above we simply use Cancel to represent either possibility.

The overall transaction which glues everything together is as follows:

$$\begin{array}{lcl} \mathit{Transaction} & \stackrel{\text{def}}{=} & \mathbf{t}(\mathit{Shop}\,;\mathit{Offers}\,;(\mathit{Packing} \mid \mathit{Bank})\,;\mathit{Couriers}, \\ & & \quad \mathit{Operator}, \mathit{done}, \mathit{done}) \end{array}$$

4.2 SOCK

SOCK [GLG+06, GLMZ08, LZ09] is aimed specifically as a calculus for service oriented computing. It has three layers intended for modelling service oriented architectures. The layers are the *service behaviour calculus*, the *service engine calculus*, and the

services system calculus. The first layer describes the low-level behaviour of a service; the second layer has the aim of handling the various instantiations of a service; while the third layer allows for the definition of the whole system, possibly including various service engines.

In this overview, we will focus on the behaviour layer. SOCK has a number of specialised constructs intended for modelling the service-oriented architecture. For example, SOCK provides the request-response mode of communication. This differs from the standard input-output communication in that a client can solicit an activity on a server and receive the output of the activity back. For such a scenario, SOCK also offers other notions such as the concept of a location. A process is not only distinguished by its definition but also on the location where it is running.

SOCK provides a flexible error handling mechanism with three variations: fault handling, termination handling and compensation handling. The mechanism relies on a handler which is a function associating names with processes. If a fault name is thrown, the corresponding process (according to the function) is executed. The termination and compensation handling mechanism is associated to the scope. A scope can be thought of as an execution container to which a termination can be associated. If the scope terminates and the termination handler has not yet been invoked, then the termination handler becomes a compensation handler because the scope has been successfully completed. SOCK provides a high level of flexibility by allowing the modification of the fault and termination handlers to be carried out at any point of the process execution in the corresponding scope.

Another peculiarity of SOCK is that it provides a mechanism for distributed compensation. This is achieved by allowing a server to send a failure handler to the client. Thus, if the operation on the server fails, the client is informed by the server how compensation can take place.

Syntax

SOCK's syntax is given as follows, with P, Q ranging over processes, o over one-way communication operations, o_r over request-response operations, l over locations, q over scope names, f over fault names, u over scope names and fault names:

$$
\begin{array}{rcll}
P, Q & ::= & o \mid o_r(P) & \text{input} \\
& \mid & \overline{o}@l \mid \overline{o_r}@l(\mathcal{H}) & \text{output} \\
& \mid & P\,;Q & \text{sequential composition} \\
& \mid & P \mid Q & \text{parallel composition} \\
& \mid & P + Q & \text{non-deterministic choice} \\
& \mid & 0 & \text{inert process} \\
& \mid & \{P\}_q & \text{scope} \\
& \mid & \mathit{inst}(\mathcal{H}) & \text{handler update} \\
& \mid & \mathit{throw}(f) & \text{throw} \\
& \mid & \mathit{comp}(q) & \text{compensate} \\
& \mid & cH & \text{current handler}
\end{array}
$$

A SOCK process can be a one-way (o) or a request-response input, written as $o_r(P)$, where P is the process to be executed upon an incoming request; a one-way ($\overline{o}@l$) or a request response output, written as $\overline{o_r}@l(\mathcal{H})$, such that $\mathcal{H}$ is the handler to be used in case the external operation requested fails; a sequential or a parallel composition; a non-deterministic choice, denoted by $P + Q$; the inert process; a scope-bounded process, denoted by $\{P\}_q$, with P as its operation and q as its name; the installation of a handler $\mathcal{H}$, denoted by $\mathit{inst}(\mathcal{H})$; the throw of a fault f, denoted by $\mathit{throw}(f)$; the activation of compensation for a scope name q, written as $\mathit{inst}(q)$; or a reference to the current handler, cH.

To facilitate the definition of the semantics, the following additional syntax is required:

$$
\begin{array}{rcll}
P, Q & ::= & \mathit{Exec}(P, o_r, l) & \text{request-response execution} \\
& \mid & \{P : \mathcal{H} : u\}_{q_\perp} & \text{active scope} \\
& \mid & o_r(\mathcal{H}) & \text{receive for response} \\
& \mid & o_r\langle\mathcal{H}\rangle & \text{dead receive for response} \\
& \mid & \overline{o_r}!f@l & \text{fault send}
\end{array}
$$

A process can be an executing request-response $\mathit{Exec}(P, o_r, l)$ such that P is the process being executed, o_r the invoking operation and l the location; an active scope $\{P : \mathcal{H} : u\}_{q_\perp}$ where P is the running process, $\mathcal{H}$ the fault handler, u the name of a handler which is waiting for the appropriate time to be executed, and $q_\perp$ stands for either the name of the scope or $\perp$ (representing the fact that the scope is in zombie state); receiving a response for an earlier outgoing request-response, denoted by $o_r(\mathcal{H})$; receiving a response for an earlier outgoing request-response while in zombie state, denoted by $o_r\langle\mathcal{H}\rangle$; or the sending of a fault f, $\overline{o_r}!f@l$, in response to an earlier request.

Semantics

The semantics of SOCK is given in terms of a considerable number of rules. For this overview we will focus on some important ones. The first four rules give some insight regarding the request-response mechanism:

$$o_r(P) \xrightarrow{\uparrow o_r@l} Exec(P, o_r, l) \qquad Exec(0, o_r, l) \xrightarrow{\downarrow \overline{o_r}@l} 0$$

$$\overline{o_r}@l(\mathcal{H}) \xrightarrow{\uparrow \overline{o_r}@l} o_r(\mathcal{H}) \qquad o_r(\mathcal{H}) \xrightarrow{\downarrow o_r} inst(\mathcal{H})$$

The upper two rules deal with the processing of an incoming request-response: upon a request ($\uparrow o_r@l$), the execution of process P is initiated; once P completes, the response ($\downarrow \overline{o_r}@l$) is sent back to the requester. The lower two rules deal with an outgoing request-response: A request is first sent to a server and a response is expected; as soon as the response arrives back, the handler is installed.

Consider the following rules concerning the activation of a fault handler and the installation of compensation respectively:

$$\{0 : \mathcal{H} : f\}_{q_\perp} \xrightarrow{\tau} \{\mathcal{H}(f) : \mathcal{H} \oplus [f \mapsto \perp] : \perp\}_{q_\perp} \qquad \{0 : \mathcal{H} : \perp\}_q \xrightarrow{inst(cmp(\mathcal{H}))} 0$$

The first rule states that if a name f is waiting to be executed in the scope, its handler is loaded (the subsequent two rules ensure that f always has a handler) and removed from the handler function $\mathcal{H}$. The second rule installs the scope termination handler ($cmp(\mathcal{H})$) to the next higher scope so that this can act as compensation. Note that the main activity within the scope is the inert process, meaning that the activity has successfully terminated.

Next, we consider what happens when a fault is thrown:

$$\frac{P \xrightarrow{th(f)} P', \mathcal{H}(f) \neq \perp}{\{P : \mathcal{H} : u\}_{q_\perp} \xrightarrow{\tau} \{P' : \mathcal{H} : f\}_{q_\perp}} \qquad \frac{P \xrightarrow{th(f)} P', \mathcal{H}(f) = \perp}{\{P : \mathcal{H} : u\}_q \xrightarrow{th(f)} \langle \{P' : \mathcal{H} : \perp\}_\perp \rangle}$$

The first rule deals with the case when a fault handler is available, i.e. $\mathcal{H}(f) \neq \perp$. In such a case, the fault name is set in the scope waiting to be executed. On the other hand, if no handler is available, the scope changes into zombie mode — marked with the scope name $\perp$ — and the fault is rethrown to the next outer scope. A scope in zombie mode signifies that the scope has not completed successfully. However, a zombie scope is still allowed to continue executing a number of particular processes

controlled by the *killable* function. This function kills all processes which are in parallel to an unhandled fault (forced termination) but allows mainly two kinds of processes to continue: pending communication and termination handlers of forcedly terminated child scopes. Note that the zombie scope is enclosed in angled brackets signifying protection, indicating that the scope can no longer be killed by external faults.

Another important detail is that to ensure that the handler updates are executed before a fault is handled, execution of a fault throw is not allowed to take place unless the function *killable* is defined. Given that this function is undefined on handler installations, fault throws will have to wait for handler updates to take place.

Finally, the next set of rules deal with installing and discarding compensations:

$$\frac{P \xrightarrow{inst(\mathcal{H})} P'}{\{P : \mathcal{H}' : u\}_{q_\perp} \xrightarrow{\tau^*} \{P' : \mathcal{H}' \oplus \mathcal{H} : u\}_{q_\perp}}$$

$$\frac{P \xrightarrow{cm(q,Q)} P', \mathcal{H}(q) = Q}{\{P : \mathcal{H} : u\}_{q'_\perp} \xrightarrow{\tau} \{P' : \mathcal{H} \oplus [q \mapsto 0] : u\}_{q'_\perp}}$$

In the first rule, a handler ($\mathcal{H}$) is installed. Recall that a handler is a function from names to processes. Thus, the operator $\oplus$ merges the two functions $\mathcal{H}$ and $\mathcal{H}'$ into a single function, replacing any conflicts by the mapping of the new handler ($\mathcal{H}$). The second rule states that after the execution of compensation Q, the corresponding handler should be removed, i.e. the function mapping the name q is set to the inert process.

Example

The example encoded in SOCK is presented below starting from the definition of the basic activities:

$$\begin{aligned} \mathit{Store} &\stackrel{\text{def}}{=} \mathit{order}\,;(\overline{x}@ST \mid (x) + (x\,;\mathit{throw}(st)) \\ \mathit{Order} &\stackrel{\text{def}}{=} \overline{\mathit{order}}@ST([s \mapsto (cH \mid \mathit{ReStock} \mid \mathit{Email}), st \mapsto \mathit{throw}(f)]) \\ \mathit{ReStock} &\stackrel{\text{def}}{=} \overline{\mathit{restock}}@ST([rs \mapsto \mathit{throw}(g)]) \end{aligned}$$

Modelled in a service-oriented fashion, the *Order* activity contacts the *Store* through channel *order*. If the activity at the *Store* succeeds, then the activity exits, otherwise, the activity throws fault *st*. This fault is handled by *Order* and rethrown as fault f which is in turn handled by the main transaction (given later) — triggering compensation for scope s. The activities should be all modelled in this fashion. A

slight difference applies to compensating activities where the fault to be triggered is g (instead of f) as in the case of *ReStock*. This causes the *Operator* activity to start. Finally, note that the handler named s will be used to accumulate compensations in parallel — hence the use of cH which refers to the currently installed compensation.

Next, we give the definition of processes related to the offers:

$$\begin{array}{rcl}
\mathit{Offer}_1 & \stackrel{\text{def}}{=} & \overline{\mathit{offer}_1}@SP([\mathit{failedOffer} \mapsto \mathit{Offer}_2]) \\
\mathit{Offer}_2 & \stackrel{\text{def}}{=} & \overline{\mathit{offer}_2}@SP \\
\mathit{Offers}_1 & \stackrel{\text{def}}{=} & ((\mathit{offer}_1\,;(\overline{x}@MK \mid (x\,; \mathit{inst}([s \mapsto (cH \mid \mathit{Withdraw})])))) + \\
 & & \quad (x\,; \mathit{throw}(\mathit{failedOffer}))) \\
\mathit{Offers}_2 & \stackrel{\text{def}}{=} & \mathit{offer}_2\,;(\overline{x}@MK \mid (x\,; \mathit{inst}([s \mapsto (cH \mid \mathit{Withdraw})])) + (x))
\end{array}$$

If the first offer fails, the exception *failedOffer* is thrown. However, this is caught by Offer_1 and is not propagated further. *failedOffer* is handled by attempting Offer_2. If this fails as well, no action is taken. On the other hand, if either of the offers succeed, the compensation *Withdraw* is installed.

Finally, we consider the courier booking:

$$\begin{array}{rcl}
\mathit{Courier} & \stackrel{\text{def}}{=} & \{\overline{\mathit{courier}_1}@C_1([c_1 \mapsto \mathit{Cancel}_1, \mathit{failedCourier}_1 \mapsto \overline{\mathit{failed}_1}@C])\,; \\
 & & \quad \overline{\mathit{booked}_1}@C\}_{c1} \mid \\
 & & \{\overline{\mathit{courier}_2}@C_2([c_2 \mapsto \mathit{Cancel}_2, \mathit{failedCourier}_2 \mapsto \overline{\mathit{failed}_2}@C])\,; \\
 & & \quad \overline{\mathit{booked}_2}@C\}_{c_2} \mid \\
 & & (\ (\mathit{booked}_1\,;(\mathit{failed}_2 + (\mathit{booked}_2\,; \mathit{comp}(c_2))\,; \\
 & & \quad\ \mathit{inst}([s \mapsto (cH \mid \mathit{Cancel}_1)]))) + \\
 & & \quad (\mathit{failed}_1\,;(\mathit{booked}_2\,; \mathit{inst}([s \mapsto (cH \mid \mathit{Cancel}_2)]) + \\
 & & \quad\ (\mathit{failed}_2\,; \mathit{throw}(f)))))
\end{array}$$

For the *Courier* process, the non-deterministic choice has been used to distinguish among the possible outcomes. If both succeed, then the second courier is compensated by running the compensation associated to the scope ($\mathit{comp}(c_2)$). If either of them fails and the other succeeds, then the appropriate compensation is added to the higher compensation scope (s). Finally, if both fail, then the fault f is thrown, causing the whole transaction to fail.

We end this example by giving the topmost transaction which refers to the above defined processes:

$$\begin{array}{rcl}
\mathit{Transaction} & \stackrel{\text{def}}{=} & \{\mathit{inst}([f \mapsto \mathit{comp}(s), g \mapsto \mathit{Operator}]) \mid \\
 & & \quad (\mathit{Order}\,; \mathit{Offer}_1\,;(\mathit{Pack} \mid \mathit{Credit})\,; \mathit{Courier})\}_s
\end{array}$$

Note the installation of the fault handlers f and g, the former initiating the compensation of scope s.

4.3 webπ

webπ [LZ05, MG05] is an extension of asynchronous π calculus with a special process for handling transactions. A notable aspect of webπ is that it has the notion of a timeout; if a transaction does not complete within a given number of cycles, then it is considered to have failed. A variation of webπ which abstracts from this notion of time is webπ_∞ [ML06]. When the notion of time is abstracted, a transaction which has not timed out is a transaction with normal behaviour, while timed out is a failure. Both flavours of webπ provide the notion of mobility which is important when modelling web services. For this reason, machines are a layer on top of processes such that different machines with different processes can be executing concurrently. However note that time is not synchronised across machines, but only between parallel processes on the same machine. In this overview, we will focus on webπ_∞ and hence we will not deal with time. Furthermore, we choose also to leave out machines.

Syntax

The syntax of webπ_∞ is as follows, with P, Q ranging over processes, x, y, z, u ranging over channel names, $\widetilde{u}$ ranging over tuples of names, and i over a finite non-empty set I of indexes:

P	::=	0	nil
	|	$\overline{x}\,\widetilde{u}$	output
	|	$\Sigma_{i\in I} x_i(\widetilde{u_i}).P_i$	guarded choice
	|	$(x)P$	restriction
	|	$P \mid P$	parallel composition
	|	$P \oplus P$	internal choice
	|	$!x(\widetilde{u}).P$	guarded replication
	|	$\langle\!\langle\, P\,;P \,\rangle\!\rangle_x$	work unit

A process can be the inert process; $\overline{x}\,\widetilde{u}$, denoting an output of a tuple of names u on a channel x; an input-guarded choice $\Sigma_{i\in I} x_i(\widetilde{u_i}).P_i$ such that upon an input x_i, the process evolves to P_i; a process P with a restriction on a name x, written as $(x)P$; a parallel composition; an internal choice written as $P \oplus P$; an input guarded replication $!x(\widetilde{u}).P$, spawning a process P upon an input on x; or a work unit $\langle\!\langle\, P\,;Q \,\rangle\!\rangle_x$ which behaves as P until an abort is signalled on $\overline{x}$, after which it behaves as Q.

Semantics

It is evident from the syntax that $\texttt{web}\pi_\infty$ does not provide a direct construct for compensation. The work unit construct available is closer to the concept of exception handling rather than a compensation approach. However, the work unit is still expressive enough to encode compensations.

The semantics of the calculus is given in terms of a structural congruence and a reduction relation. In what follows, we give a description of the important aspects of $\texttt{web}\pi_\infty$ while highlighting the relevant rules. Consider the following two axioms:

$$\langle\!|\, 0\,;Q \,|\!\rangle_x \equiv 0 \qquad \langle\!|\, \langle\!|\, P\,;Q \,|\!\rangle_y \mid R\,;S \,|\!\rangle_x \equiv \langle\!|\, P\,;Q \,|\!\rangle_y \mid \langle\!|\, R\,;S \,|\!\rangle_x$$

The first axiom states that upon the successful completion of a work unit, the reparation Q is discarded. In other words this means that a transaction cannot be compensated once it has completed. The second axiom shows how nested transactions are flattened. This implies that the failure of a parent transaction does not affect its children and vice-versa (unless explicitly programmed).

There are another two axioms which deal with work units. Their purpose is to allow restrictions and outbound messages to float in an out of work unit boundaries in a similar fashion to nested transactions. The rules are below:

$$\begin{array}{rcl} \langle\!|\, (z)P\,;Q \,|\!\rangle_x & \equiv & (z)\langle\!|\, P\,;Q \,|\!\rangle_x \;\; \textit{if} \;\; z \notin \{x\} \cup \textit{fn}(Q) \\ \langle\!|\, \overline{z}\,\widetilde{u} \mid P\,;Q \,|\!\rangle_x & \equiv & \overline{z}\,\widetilde{u} \mid \langle\!|\, P\,;Q \,|\!\rangle_x \end{array}$$

Note that in the first axiom, if z is the name of the work unit ($z = x$), then taking the restriction outside of the unit would render the unit unabortable from the outside. On the other hand, if z is a free name in Q ($z \in \textit{fn}(Q)$), z in Q would be bound if the restriction of z is taken outside the unit.

The rule responsible for aborting a unit (not given here),upon receiving an abort signal (an output on the name of the unit), replaces the main body of the process by the exception handler. This has the purpose of simultaneously aborting the main body and giving control to the exception handler. Note that the exception handler is protected from any further external abort signals by restricting the name of the new work unit using the first axiom above. Purposely, the rule does not fire if the work unit includes any nested transactions, outbound communication or restrictions, as these must first float out of the work unit, before the work unit can be aborted.

Example

The example encoded in $\texttt{web}\pi_\infty$ is given below, starting with the basic activities:

$$
\begin{array}{rcl}
\mathit{Order} & \stackrel{\text{def}}{=} & \overline{s_1} \oplus \overline{f_1} \\
\mathit{Restock} & \stackrel{\text{def}}{=} & 0 \oplus \overline{g} \\
\mathit{ReStock} & \stackrel{\text{def}}{=} & \langle\!\langle\ \mathit{Order} \mid s_1.c_1.\overline{w_1}\,;\mathit{ReStock} \mid \mathit{Email}\ \rangle\!\rangle_{w_1} \\
\mathit{Pack} & \stackrel{\text{def}}{=} & \langle\!\langle\ \mathit{DoPacking} \mid s_3.c_3.\overline{w_3}\,;\mathit{Unpack}\ \rangle\!\rangle_{w_3} \\
\mathit{Credit} & \stackrel{\text{def}}{=} & \langle\!\langle\ \mathit{DoCrediting} \mid s_4.c_4.\overline{w_4}\,;\mathit{Refund}\ \rangle\!\rangle_{w_4}
\end{array}
$$

The possibility of a success or failure of an activity is modelled through internal choice, signifying a success by an output on s_i or a failure on f_i. A central issue in this example is that the compensable actions should only be activatable once the corresponding action has been completed successfully. For this reason, in parallel with each action, the channel input s_i is used to wait for the signal of the successful completion of the activity. Unless s_i is received, c_i cannot accept any input. For this reason, the failure of any activity is always triggered through c_i (which in turn triggers w_i) and never is the name of work unit (w_i) directly invoked. The other activities should be encoded in a similar fashion. In the case of compensating activities, their failure triggers a fault g which is handled (later on) by activating the *Operator* activity.

Another important issue is that transactions with the possibility of compensating after completion should be carefully treated since (in $\mathtt{web}\pi_\infty$) the compensation of a completed transactions (work units) is discarded. This is the reason behind the presence of c_i — c_i waits for input and only after c_i is received, is the reparation triggered (output on w_i).

Next, we consider the encoding of the offers:

$$
\begin{array}{rcl}
\mathit{Offer}_1 & \stackrel{\text{def}}{=} & \overline{s_2} \oplus \overline{x} \\
\mathit{Offer}_2 & \stackrel{\text{def}}{=} & \overline{s_2} \oplus 0 \\
\mathit{Offers}_1 & \stackrel{\text{def}}{=} & \langle\!\langle\ \mathit{Offers}_2 \mid s_2.c_2.\overline{w_2}\,;\mathit{Withdraw}\ \rangle\!\rangle_{w_2} \\
\mathit{Offers}_2 & \stackrel{\text{def}}{=} & \langle\!\langle\ \mathit{Offer}_1\,;\mathit{Offer}_2\ \rangle\!\rangle_{x}
\end{array}
$$

In case Offer_1 fails, it simply signals failure to the work unit in Offers_2. This, in turn, triggers Offer_2. If any of these succeed, the compensation *Withdraw* is activated by output on s_2. However, if both fail, nothing happens, i.e. the main transaction is not affected.

The encoding of the courier booking is given below:

$$
\begin{array}{rcl}
\mathit{Book}_1 & \stackrel{\text{def}}{=} & (\overline{s'_1} \mid \overline{s_5}) \oplus \overline{f'_1} \\
\mathit{Book}_2 & \stackrel{\text{def}}{=} & (\overline{s'_2} \mid \overline{s_6}) \oplus \overline{f'_2} \\
\mathit{Courier}_1 & \stackrel{\text{def}}{=} & \langle\!| \ \mathit{Book}_1 \mid s_5.c_5.\overline{w_5} \,;\, \mathit{Cancel}_1 \ |\!\rangle_{w_5} \\
\mathit{Courier}_2 & \stackrel{\text{def}}{=} & \langle\!| \ \mathit{Book}_2 \mid s_6.c_6.\overline{w_6} \,;\, \mathit{Cancel}_2 \ |\!\rangle_{w_6} \\
\mathit{Couriers} & \stackrel{\text{def}}{=} & \mathit{Courier}_1 \mid \mathit{Courier}_2 \mid (s'_1.(s'_2.\overline{c_6} + f'_2) + f'_1.(s'_2 + f'_2.\overline{f_5}))
\end{array}
$$

The case of the courier is handled among five processes. The *Couriers* process starts the booking of $\mathit{Courier}_1$ and $\mathit{Courier}_2$ in parallel. The booking is then handled by the two processes Book_1 and Book_2. If a booking succeeds, both *Couriers* and $\mathit{Courier}_1$ (or $\mathit{Courier}_2$) are notified: $\mathit{Courier}_1$ is notified so that the compensation is made available by exposing c_5 (or c_6 in the case of $\mathit{Courier}_2$). *Couriers* has to also be notified in order to take the necessary decision regarding both bookings. If both succeed, one of them is compensated; if both fail, then a failure is signalled to the main transaction (*Transaction*); otherwise, if one succeeds and the other fails, no action is taken.

Finally, we give the definition of the main transaction as follows:

$$
\begin{array}{rcl}
\mathit{Transaction} & \stackrel{\text{def}}{=} & \langle\!| \ \mathit{ReStock} \mid \mathit{Offers} \mid \mathit{Pack} \mid \mathit{Credit} \mid \mathit{Couriers} \mid \\
& & \quad \sum_{i \in \{1..5\}} f_i.\overline{\mathit{main}} \,; \\
& & \quad \langle\!| \ \overline{c_1} \mid \overline{c_2} \mid \overline{c_3} \mid \overline{c_4} \mid \overline{c_5} \mid \overline{c_6} \,;\, \mathit{Operator} \ |\!\rangle_h \ |\!\rangle_{\mathit{main}}
\end{array}
$$

Note that *Transaction* starts all the main processes in parallel (diverting from the original specification of the example) since $\mathtt{web}\pi_\infty$ does not provide a sequential composition operator.

4.4 dcπ

dcπ [VFR09] tries to address various limitations of its predecessors. In particular, it supports dynamic construction of compensations; guarantees that installations and activations of compensations take place (through proofs); and preserves compensations till after the completion of transactions so that completed transactions can also be compensated for.

A central aspect of dcπ is that it assigns a unique identifier to each transaction. It is through this identifier that a transaction can be signalled to start executing its compensation. For this reason any process which is allowed to signal a transaction t to compensate, should be given access to channel t (whose name corresponds to the identifier of the transaction). Scope extrusion is in fact heavily heavily used in

dcπ for this purpose. Informally, scope extrusion occurs when a restricted (local) channel is passed on to another process (outside of the scope of the channel). This technique is also used to pass control among the processes and thus allowing a form of sequentiality which is not explicitly available in dcπ.

Syntax

dcπ has two kinds of processes: *compensable processes* and *execution processes*. In addition to compensable processes, execution processes can also include stored compensations. The syntax of compensable processes is given next with i ranging over the natural numbers; P, Q, P_i and Q_i ranging over compensable processes; a and a_i ranging over a countable set of channel names N; t ranging over a countable set of transaction identifiers T; v and x ranging over the union of the disjoint sets N and T; and $\tilde{v}$ and $\tilde{x}$ ranging over sequences of identifiers.

P, Q	$::=$	0	inaction
	$\mid$	$\overline{t}$	failure
	$\mid$	$\overline{a}\langle\tilde{v}\rangle$	output
	$\mid$	$\Sigma_{i\in I} a_i(\tilde{x}_i)\%Q_i.P_i$	input guarded choice
	$\mid$	$!a(\tilde{x})\%Q.P$	input guarded replication
	$\mid$	$(P \mid Q)$	parallel composition
	$\mid$	$(\nu\ x)P$	restriction
	$\mid$	$\langle P\rangle$	protected block
	$\mid$	$t[E]$	transaction scope

Thus, a compensable process may be the inert action; a failure signal $\overline{t}$ for a transaction named t; an output $\tilde{v}$ on a channel a, written as $\overline{a}\langle\tilde{v}\rangle$; an input guarded choice $\Sigma_{i\in I} a_i(\tilde{x}_i)\%Q_i.P_i$ where upon an input on a_i, the process evolves to P_i, installing compensation Q_i; a replication $!a(\tilde{x})\%Q.P$, spawning process P and installing compensation Q upon each input on a; scope restriction $(\nu\ x)P$, restricting x in P; a protected block $\langle P\rangle$ (explained below); or a transaction scope $t[E]$ named t with body E (whose syntax is given below).

The syntax of an execution process is given below with E and F ranging over execution processes:

E, F	$::=$	P	compensable process
	$\mid$	$\{P\}$	stored compensation
	$\mid$	$(E \mid F)$	parallel composition
	$\mid$	$(\nu\ x)E$	restriction

An execution process may be a compensable process; a stored compensation $\{P\}$ with

body P; a parallel composition of execution processes; or a scope restriction $(\nu\ x)E$.

Semantics

An aspect which is particular to $\mathsf{dc}\pi$ is that compensation is associated to input rather than a successfully completed action. Note that compensations are associated to input and not output because output can be used to signal failure. The advantage of this approach is that it offers a high level of dynamicity in the installation of compensations such that a compensation process can be composed in parallel to the currently stored compensation upon each input. This logic is formally given in terms of reduction rules on contexts. Consider the following two rules where D is a double context (two comma-separated contexts running in parallel):

$$\frac{\begin{array}{c}D \text{ does not bind } a \\ a = a_j \text{ for some } j \in I\end{array}}{D[\![\overline{a}\langle\tilde{v}\rangle, \Sigma_{i\in I} a_i(\tilde{x}_i)\%Q_i.P_i]\!] \to D[\![0, \{Q_j\{^{\tilde{v}}/_{\tilde{x}_j}\}\} \mid P_j\{^{\tilde{v}}/_{\tilde{x}_j}\}]\!]}$$

$$\frac{D \text{ does not bind } a}{D[\![\overline{a}\langle\tilde{v}\rangle, !a(\tilde{x})\%Q.P]\!] \to D[\![0, \{Q\{^{\tilde{v}}/_{\tilde{x}}\}\} \mid P\{^{\tilde{v}}/_{\tilde{x}}\} \mid !a(\tilde{x})\%Q.P]\!]}$$

The first rule handles the input guarded choice such that upon taking one of the options, the associated compensation Q_i is installed as a stored compensation (within curly brackets) in parallel to the main body of the transaction. The second rule is similar but handles the input guarded replication instead. Note that compensations are always installed in parallel to other compensations in $\mathsf{dc}\pi$ (no matter the order in which they were originally executed).

After considering how compensations are installed, we consider how compensations are activated. In $\mathsf{dc}\pi$ compensations are automatically activated upon a failure signal. A failure is signalled with an output on the identifier of the transaction. Once this signal is received, the failed transaction is discarded while the stored compensations are "extracted" and put inside a protected block. The extraction of the stored compensations is handled by the function extr which, given an execution process, returns a compensation process. The following two reduction rules explain what happens upon an internal error and an external error respectively:

$$\frac{C \text{ does not bind } t}{t[C[\![\ \bar{t}\ ]\!]] \to \mathsf{extr}(C[\![0]\!])} \qquad \frac{C \text{ does not bind } t}{t[D[\![\ \bar{t}, t[E]]\!]] \to D[\![0, \mathsf{extr}(E)]\!]}$$

In the case of the first rule, the failure signal is invoked from within the transaction. Note that if the context C binds the name of the transaction, the invocation will

not reach the transaction. As soon as failure reaches the transaction, the function extr is invoked on the outer context in order to discard the remainder of the transaction (including sub-transactions) and extract the stored compensations, putting them in protected blocks. Thus the function accepts an execution process (possibly with stored compensations) and returns a compensable process (with no stored compensations). Note that protected blocks and scoping constructs are not discarded by extr.

Finally, in dcπ compensations are not discarded upon the termination of the corresponding transaction, allowing transactions to be compensated even after they complete.

Example

dcπ relies heavily on channel communication both for programming sequences of processes and for programming compensations. For this reason the example has been programmed in terms of a number of communicating entities with the initial request originating from the client. The example encoded in dcπ is given below, starting with the basic activities:

$$\begin{array}{rcl} \mathit{Order} & \stackrel{\text{def}}{=} & (\nu\ o)\ o\ [\{\overline{b}\} \mid (\nu\ x, msg)(\overline{x}.(x\%(\mathit{ReStock}).\overline{\mathit{stockOk}} + x.\overline{o}))] \\ \mathit{ReStock} & \stackrel{\text{def}}{=} & (\nu\ r)\ r\ [\{\overline{op}\} \mid (\nu\ x)(\overline{x} \mid x.0 + x.\overline{r})] \end{array}$$

We start by defining the *Order* activity, modelled as a transaction scope in which a non-deterministic choice between success or failure is modelled through an input guarded choice on x. If the order is successful, *ReStock* is installed as compensation, otherwise failure is signalled by outputting on the transaction name (in this case $\overline{o}$). Note that a statically stored compensation has been used $\{\overline{b}\}$ so that upon failure of the order, the whole transaction is signalled as failure — triggering the global compensation. The compensating activity *ReStock* is modelled similarly to *Order* with the difference that it does not have a compensation, and upon failure it outputs on *op* (through the static compensation mechanism) to trigger the *Operator* activity. The other basic activities can be modelled in a similar fashion to the above.

However, a failure of a process should not always cause the failure of the overall transaction — this is the case of the offers below:

$$\begin{array}{rcl} \mathit{Offer}_1 & \stackrel{\text{def}}{=} & (\nu\ h)\ h\ [(\nu\ x, msg)(\{\mathit{Offer}_2\} \mid \overline{x} \mid x\%(\mathit{Withdraw}).0 + x.\overline{h})] \\ \mathit{Offer}_2 & \stackrel{\text{def}}{=} & (\nu\ i)\ i\ [(\nu\ x, msg)(\overline{x} \mid x\%(\mathit{Withdraw}).0 + x.0))] \end{array}$$

In the case of Offer_1, the static compensation is not the failure of whole transaction but the execution of Offer_2, i.e. $\{\mathit{Offer}_2\}$ instead of $\{\overline{b}\}$. Offer_2, in turn, does not

have any compensation specified. Thus in case of failure, this will have no effect on the overall transaction.

Another interesting process is the booking of the courier:

$$
\begin{array}{rcl}
\mathit{Courier}_1 & \stackrel{\text{def}}{=} & (\nu\ k)\ k\ [(\nu\ x)(\overline{x} \mid x\%(\mathit{Cancel}_1).\overline{\mathit{booked}_1} + x.\overline{\mathit{failed}_1}.\overline{k}))] \\
\mathit{Courier}_2 & \stackrel{\text{def}}{=} & (\nu\ l)\ l\ [(\nu\ x)(\overline{x} \mid x\%(\mathit{Cancel}_2).\overline{\mathit{booked}_2} + z.\overline{\mathit{failed}_2}.\overline{l})] \\
\mathit{Couriers} & \stackrel{\text{def}}{=} & (\nu\ j)\ j\ [\{\overline{b}\} \mid \mathit{Courier}_1 \mid \mathit{Courier}_2 \mid \\
 & & \quad (\mathit{booked}_1.(\mathit{failed}_2.\overline{\mathit{courierOk}} + \mathit{booked}_2.\overline{l}.\overline{\mathit{courierOk}}) + \\
 & & \quad (\mathit{failed}_1.(\mathit{booked}_2.\overline{\mathit{courierOk}} + \mathit{failed}_2.\overline{j})))]
\end{array}
$$

In this case, the booking of two couriers is initiated in parallel while the process *Couriers* waits for the outcomes. As soon as the outcomes are available, using the input guarded choice, *Couriers* decides whether to cancel one of the couriers or send a failure signal to the *Transaction* process in case both fail.

Finally, we specify the overall transaction and the client interaction below:

$$
\begin{array}{rcl}
\mathit{Client} & \stackrel{\text{def}}{=} & (\nu\ a)\ a\ [(\nu\ \mathit{ctl}, \mathit{msg})(\overline{\mathit{ord}}\langle \mathit{ctl}, \mathit{msg}\rangle \mid \mathit{ctl}(b)\%(\mathit{msg}(\mathit{failed})) \mid \\
 & & \quad (\overline{\mathit{ctl}}\langle a\rangle \mid (\nu x)(\overline{x} \mid (x.\overline{b} \mid x.(\mathit{msg}(\mathit{done})))))))] \\
\mathit{Transaction} & \stackrel{\text{def}}{=} & !\mathit{ord}(\mathit{ctl}, \mathit{msg}).(\nu\ b)\ b\ [\overline{\mathit{ctl}}\langle b\rangle \mid \mathit{ctl}(a)\%(\overline{a} \mid \overline{\mathit{msg}}\langle \mathit{failed}\rangle) \\
 & & \quad (\nu\ \mathit{stockOk}, \mathit{bankOk}, \mathit{packOk}, \mathit{courierOk}) \\
 & & \quad (\mathit{Order} \mid \mathit{Offer}_1 \mid \mathit{Pack} \mid \mathit{Charge} \mid \mathit{Couriers} \mid \\
 & & \quad \mathit{op}.\mathit{Operator} \mid \\
 & & \quad \mathit{stockOk}.\mathit{bankOk}.\mathit{packOk}.\mathit{courierOk}.\overline{\mathit{msg}}\langle \mathit{done}\rangle)]
\end{array}
$$

As soon as the client issues an order (on channel *ord*), the shop responds by initiating a number of processes in parallel: *Order*, Offer_1, *Pack*, *Charge*, and *Couriers*. Note that the client can decide to cancel the order at any time. This is programmed using a non-deterministic choice between issuing a failure signal $\overline{b}$ on the name of the main transaction (b has been passed to the client by scope extrusion), and waiting for the *done* message to be received. This definition of processes diverts from the original specification in that all the processes are done in parallel. It is possible to keep to the original specification, but the definition in $\mathsf{dc}\pi$ becomes cluttered. If the *Transaction* process fails, the client transaction is terminated with failure and a message indicating failure is sent to the client (representing an email notification). Finally, recall that according to the specification, if the transaction fails the operator should be notified. This has been modelled by the *Operator* activity waiting for an input on *op*.

4.5 COWS

COWS [LPT07, LPT08a] is specifically targeted to formally model web services, providing a way of applying formal methods to web services. In COWS, an endpoint which receives or sends invocations is a tuple: a partner and an operation. Thus, each partner can be involved in various concurrent operations. A partner can be thought of as a location. Notably, COWS provides a very restricted basic syntax, using which, richer syntax is defined. Particularly, no notion of failure, success, transaction or scope is provided in the basic syntax. Note that in COWS, a computational entity is called a *service* rather than a process.

Syntax

The syntax of COWS is given below with s ranging over services; k ranging over killer labels; w ranging over variables and values ($\bar{w}$ over tuples); e ranging over expressions ($\bar{e}$ over tuples); d ranging over killer labels, names and variables; p ranging over names used to refer to partners; o ranging over names used to refer to operations; and u, u' ranging over names and variables:

$$\begin{array}{lcll} g & ::= & 0 \mid p \cdot o\,?\,\bar{w}.s \mid g + g & \text{input-guarded choice} \\ s & ::= & \mathbf{kill}(k) \mid u \cdot u'\,!\,\bar{e} \mid g \mid s|s \mid \{\!|s|\!\} \mid [d]\ s \mid * s & \text{services} \end{array}$$

An input-guarded choice is either the inert service 0; the receive activity $p \cdot o\,?\,\bar{w}.s$, receiving $\bar{w}$ on endpoint $p \cdot o$ and continuing as s; and the choice of two input-guarded choices, denoted by $g + g$.

A service is either a kill activity $\mathbf{kill}(k)$, representing a termination request of k's scope; an invoke activity $u \cdot u'\,!\,\bar{e}$, sending $\bar{e}$ through endpoint $u \cdot u'$ where u stands for the partner and u' stands for the operation; an input-guarded choice; a parallel composition of services $s \mid s$; a protection $\{\!|s|\!\}$, protecting s from being externally killed; a delimitation $[d]\ s$, binding d inside s; or a replication of a service $* s$.

Semantics

The semantics of COWS is given in terms of a structural congruence and a labelled transition relation. The labelled transition relation $\xrightarrow{\alpha}$ is the least relation over services induced by a number of rules of which we only present a selection here. The label α is generated by the following grammar:

$$\alpha \quad ::= \quad \dagger k \mid (p \cdot 0) \lhd \bar{v} \mid (p \cdot o) \rhd \bar{w} \mid p \cdot o)\lfloor\sigma\rfloor\bar{w}\bar{v} \mid \dagger$$

$\dagger k$ denotes the execution of a request for terminating a term k; $(p \cdot 0) \lhd \bar{v}$ and $(p \cdot o) \rhd \bar{w}$ denote an invoke and a receive action on endpoint $(p \cdot o)$ respectively; $(p \cdot o)\lfloor\sigma\rfloor\bar{w}\bar{v}$

denotes communication over $p \cdot o$ — receiving parameters $\bar{w}$ with corresponding values $\bar{v}$ with substitutions σ (still to take place); and † denotes forced termination.

We start the semantics overview by giving the rules related to killing:

$$\mathbf{kill}(k) \xrightarrow{\dagger k} 0 \qquad \frac{s_1 \xrightarrow{\dagger k} s_1'}{s_1 \mid s_2 \xrightarrow{\dagger k} s_1' \mid \mathit{halt}(s_2)} \qquad \frac{s \xrightarrow{\dagger k} s'}{[k]\, s \xrightarrow{\dagger} [k]s'}$$

The first rule specifies that a kill activity becomes the inert service 0 with a $\dagger k$ activity. The kill activity is propagated to the parallel siblings (by the second rule) resulting in the application of the *halt* function. The halt function returns the inert process for any service unless the service is a protected service, in which case the service is returned intact. The kill activity, however, is stopped as soon as a delimitation $[k]$ is encountered. This is the reason for the third rule (changing $\dagger k$ to †). Kill features yet in another rule. The rule given below ensures that if a kill is active, it takes precedence over other processes, i.e. if a kill action is active, the action is either a † or a $\dagger k$:

$$\frac{s \xrightarrow{\alpha} s' \quad d \notin \mathrm{d}(\alpha) \quad s = \mathbb{A}[\![\mathbf{kill}(d)]\!] \Rightarrow \alpha \in \{\dagger, \dagger k\}}{[d]\, s \xrightarrow{\alpha} [d]\, s'}$$

Note that $\mathbb{A}[\![\cdot]\!]$ represents a context; $\mathrm{d}(\alpha)$ returns the set of names, variables and killer labels occurring in α; and d is a name bound in s. The first condition states that α does not contain the bound name d (otherwise a special rule (not given here) takes care of the α-conversion, removing the restriction $[d]$ or in the case of $\dagger d$, the kill is stopped from propagating by the third rule above). The second condition regards the priority of kill activities, ensuring that if an activity takes place while a kill is active, it is either † (if the kill action is related to d) or $\dagger k$ if otherwise.

The constructs for handling scopes, exception handling and compensations are defined in terms of the basic syntax introduced above (Note that the sequential operator (;) is also a derived operator whose definition is not given here):

$$\begin{array}{l}\langle\!\langle [s : \mathbf{catch}(\phi_1)\{s_1\} : \ldots : \mathbf{catch}(\phi_n)\{s_n\} : s_c]_\iota \rangle\!\rangle_k = \\ \quad [p_{\phi_1}, \ldots, p_{\phi_n}](\langle\!\langle \mathbf{catch}(\phi_1)\{s_1\} \rangle\!\rangle_k \mid \ldots \mid \langle\!\langle \mathbf{catch}(\phi_n)\{s_n\} \rangle\!\rangle_k \mid \\ \qquad [k_\iota](\langle\!\langle s \rangle\!\rangle_{k_\iota} \,;\, (x_{done} \cdot o_{done}\,!\,\langle\rangle \mid \{\!|\, p_\iota \cdot o_{comp}\,?\,\langle\rangle.\langle\!\langle s_c \rangle\!\rangle_{k_\iota} \,|\!\}))) \end{array}$$

$$\begin{array}{rcl} \langle\!\langle \mathbf{catch}(\phi)\{s\} \rangle\!\rangle_k & = & p_\phi \cdot o_{fault}\,?\,\langle\rangle.[k']\langle\!\langle s \rangle\!\rangle_{k'} \\ \langle\!\langle \mathbf{undo}(\iota) \rangle\!\rangle_k & = & p_\iota \cdot o_{comp}\,!\,\langle\rangle \mid x_{done} \cdot o_{done}\,!\,\langle\rangle \\ \langle\!\langle \mathbf{throw}(\phi) \rangle\!\rangle_k & = & \{\!|\, p_\phi \cdot o_{fault}\,!\,\langle\rangle \mid x_{done} \cdot o_{done}\,!\,\langle\rangle \,|\!\} \mid \mathbf{kill}(k) \end{array}$$

The encoding $\langle\!\langle \cdot \rangle\!\rangle_k$ gives a definition of the constructs in terms of the basic COWS syntax. There are a number of important things to note: (i) A scope is made up of a service s representing normal behaviour; a number of exception handlers s_i, each associated with a particular exception signal ϕ_i and compensation handler s_c. Note that the compensation handler is protected from being killed (using $\{\!|\,|\!\}$). This is useful so that if during compensation a sibling process fails, compensation is not interrupted. Also, exception handlers are saved from being killed by throws because of the delimitation $[k_\iota]$ applied to the scope execution. (ii) A catch receives a fault signal from the respective throw and executes the exception handler in a new delimitation k' (rather than k). Thus, if the exception handler fails, it only kills the exception handler (rather than other exception handlers for example). (iii) The undo operation simply invokes the compensation through $p_\iota \cdot o_{comp}$. Note that the compensation is only listening for a trigger if the normal execution has been completed — hence the use of the sequential composition. Also note that undo is parametrised with the name of the scope (in this case ι). (iv) A throw communicates with a catch through $p_\phi \cdot o_{fault}$ and kills the scope which triggered the throw. Note that the communication with the catch is protected to prevent the kill operation from terminating this communication as well.

Example

The example encoded in COWS is given below starting with the basic activities:

$$
\begin{array}{rcl}
\mathit{Order}' & \stackrel{\text{def}}{=} & p_s \cdot o_{order}\,!\,\langle\rangle \mid p_s \cdot o_{order}\,?\,\langle\rangle.x_{done} \cdot o_{done}\,!\,\langle\rangle \mid \\
 & & \quad p_s \cdot o_{order}\,?\,\langle\rangle.\mathbf{throw}(\mathit{fault}) \\
\mathit{ReStock} & \stackrel{\text{def}}{=} & p_s \cdot o_{restock}\,!\,\langle\rangle \mid p_s \cdot o_{restock}\,?\,\langle.\rangle x_{done} \cdot o_{done}\,!\,\langle\rangle \mid \\
 & & \quad p_s \cdot o_{restock}\,?\,\langle\rangle.\mathbf{throw}(\mathit{compfault}) \\
\mathit{Order} & \stackrel{\text{def}}{=} & [\mathit{Order}' : \mathit{ReStock} \mid \mathit{Email}]_s
\end{array}
$$

The *Order* process is encoded as a scope named s with the main activity Order' and compensation *ReStock* in parallel with *Email*. The basic activities Order' and *ReStock* are similarly encoded through a non-deterministic choice in terms of an invoke in parallel with an input-guarded choice. The purpose is to model the fact that these may either succeed or fail. If they fail, they throw faults which are handled by their respective scopes (defined later). The other basic activities should follow the same pattern.

Next, we present the encoding of the offers:

$$
\begin{array}{lcl}
\mathit{Offer}_1 & \stackrel{\text{def}}{=} & p_o \cdot o_{\mathit{offer}_1}! \langle\rangle \mid p_o \cdot o_{\mathit{offer}_1}? \langle\rangle . x_{\mathit{done}} \cdot o_{\mathit{done}}! \langle\rangle \mid \\
& & \quad p_o \cdot o_{\mathit{offer}_1}? \langle\rangle . \mathit{Offer}_2 \\
\mathit{Offer}_2 & \stackrel{\text{def}}{=} & p_o \cdot o_{\mathit{offer}_2}! \langle\rangle \mid p_o \cdot o_{\mathit{offer}_2}? \langle\rangle . x_{\mathit{done}} \cdot o_{\mathit{done}}! \langle\rangle \mid \\
& & \quad p_o \cdot o_{\mathit{offer}_2}? \langle\rangle . 0 \\
\mathit{Offers} & \stackrel{\text{def}}{=} & [\mathit{Offer}_1 : \mathit{Withdraw}]_b
\end{array}
$$

In the case of the offers, the second offer is not implemented as an exception handler for the first offer. The reason is that a throw will prevent the compensation *Withdraw* from being installed. Thus, the throw-catch mechanism is bypassed and Offer_2 is called directly from Offer_1 if the latter fails. Note that no fault throw is specified in case Offer_2 fails since the failure of the offers does not affect the overall transaction.

What follows is the definition of the courier booking:

$$
\begin{array}{lcl}
\mathit{Courier}_1 & \stackrel{\text{def}}{=} & [p_c \cdot o_{\mathit{courier}_1}! \langle\rangle \mid p_c \cdot o_{\mathit{courier}_1}? \langle\rangle . x_{\mathit{done}} \cdot o_{\mathit{done}} \langle\rangle \mid \\
& & \quad p_s \cdot o_{\mathit{courier}_1}? \langle\rangle . p_c \cdot o_{\mathit{fail}_1}! \langle\rangle \mid \\
& & \quad p_c \cdot o_{\mathit{forced}_1}? \langle\rangle . \mathbf{throw}(\mathit{unhandled}) : \mathit{Cancel}_1]_{c_1} \\
\mathit{Courier}_2 & \stackrel{\text{def}}{=} & [p_c \cdot o_{\mathit{courier}_2}! \langle\rangle \mid p_c \cdot o_{\mathit{courier}_2}? \langle\rangle . x_{\mathit{done}} \cdot o_{\mathit{done}} \langle\rangle \mid \\
& & \quad p_s \cdot o_{\mathit{courier}_2}? \langle\rangle . p_c \cdot o_{\mathit{fail}_2}! \langle\rangle \mid \\
& & \quad p_c \cdot o_{\mathit{forced}_2}? \langle\rangle . \mathbf{throw}(\mathit{unhandled}) : \mathit{Cancel}_2]_{c_2} \\
\mathit{Couriers} & \stackrel{\text{def}}{=} & \mathit{Courier}_1 ; p_c \cdot o_{\mathit{forced}_2}! \langle\rangle \mid \mathit{Courier}_2 ; p_c \cdot o_{\mathit{forced}_1}! \langle\rangle \mid \\
& & \quad p_c \cdot o_{\mathit{fail}_1}? \langle\rangle . p_c \cdot o_{\mathit{fail}_2}? \langle\rangle . \mathbf{throw}(\mathit{fault})
\end{array}
$$

The speculative choice of the couriers is implemented as follows: First both courier booking requests are started, with the first one to complete terminating the other by invoking $p_c \cdot o_{\mathit{forced}_i}$. If neither booking succeeds, *Couriers* is notified on $p_c \cdot o_{\mathit{fail}_i}$ and throws an exception (*fault*) which causes the compensation of the whole transaction. Note that although a fault generated by a forcedly terminated courier is unhandled, it is still important that a fault is thrown, preventing the installation of the compensation.

Finally, the overall transaction is specified as follows:

$$
\begin{array}{lcl}
\mathit{Transaction} & \stackrel{\text{def}}{=} & [\mathit{Shop} ; \mathit{Offers} ; (\mathit{Pack} \mid \mathit{Credit}) ; \mathit{Couriers} \\
& & \quad : \mathbf{catch}(\mathit{fault})\{[\mathbf{undo}(s) \mid \mathbf{undo}(p) \mid \mathbf{undo}(b) \mid \\
& & \quad\quad \mathbf{undo}(o) \mid \mathbf{undo}(c_1) \mid \mathbf{undo}(c_2) \\
& & \quad\quad : \mathbf{catch}(\mathit{compfault})\{\mathit{Operator}\}]\} \\
& & \quad : 0]
\end{array}
$$

Note that upon the detection of fault *fault*, all the completed activities are undone

through the construct **undo**. If any of the compensating activities fail, the exception *compfault* is caught and handled by the *Operator* activity. Any compensations for sub-transactions which have not yet been completed are not executed as per the COWS definition of scope which only listens for a compensation activation after the normal scope behaviour has completed.

4.6 Committed Join

Chemical abstract machines are an attempt of emulating chemical processes as a means of computation. Similar to process algebras, chemical abstract machines are apt to model interactions among processes and how processes evolve during execution. The committed join (cJoin) [BMM04] is an extension of the join calculus [FG96], enabling it to handle compensations. The join calculus has been devised to provide an abstract foundation for object-based languages. It follows the idea of a chemical abstract machine, having a solution made up of molecules and atoms. An atom is a pending message. When atoms are joined together (composed in parallel) they become a molecule. Molecules can be heated into smaller molecules and vice-versa. For example consider:

$$\emptyset \vdash \mathit{ready}\langle \mathit{laser}\rangle, \mathit{job}\langle 1\rangle, \mathit{job}\langle 2\rangle \rightleftharpoons \emptyset \vdash \mathit{ready}\langle \mathit{laser}\rangle \mid \mathit{job}\langle 1\rangle, \mathit{job}\langle 2\rangle$$

The two atoms $\mathit{ready}\langle \mathit{laser}\rangle, \mathit{job}\langle 1\rangle$ have been glued together into the molecule $\mathit{ready}\langle \mathit{laser}\rangle \mid \mathit{job}\langle 1\rangle$. A set of molecules and atoms $\mathcal{M}$ operate within the realm of a set of reactions $\mathcal{R}$, represented by $\mathcal{R} \vdash \mathcal{M}$. A reaction $J \rhd P$ corresponds to a reduction step ($\mathcal{R} \vdash \mathcal{M} \longrightarrow \mathcal{R} \vdash \mathcal{M}'$), where any join patterns J in the solution are replaced by copies of P with replaced parameters. For example consider the following reaction D:

$$D = \mathit{ready}\langle \mathit{printer}\rangle \mid \mathit{job}\langle \mathit{file}\rangle \rhd \mathit{printer}\langle \mathit{file}\rangle$$

Applied to the set of molecules given in the previous example, we get:

$$D \vdash \mathit{ready}\langle \mathit{laser}\rangle \mid \mathit{job}\langle 1\rangle, \mathit{job}\langle 2\rangle \longrightarrow D \vdash \mathit{laser}\langle 1\rangle, \mathit{job}\langle 2\rangle$$

Finally, reactions can be created dynamically. Consider the following example where reaction D is introduced as the realm in which the molecules operate:

$$\emptyset \vdash \mathbf{def}\ D\ \mathbf{in}\ \mathit{ready}\langle \mathit{laser}\rangle \mid \mathit{job}\langle 1\rangle \mid \mathit{job}\langle 2\rangle \rightleftharpoons D \vdash \mathit{ready}\langle \mathit{laser}\rangle \mid \mathit{job}\langle 1\rangle \mid \mathit{job}\langle 2\rangle$$

Over and above the join calculus, cJoin has the following extensions: (i) a negotiation process $[P : Q]$ which behaves as P in normal execution and as Q in case of an abort; (ii) an abort signal (*abort*) which gives control to the compensation of the negotiation; (iii) a merge operator $J \blacktriangleright P$ which joins (merges) different scopes into a larger one — allowing scope messages to cross scope boundaries; (iv) a frozen molecule P is a special kind of molecules which represents a frozen compensation waiting to be activated — written as $\llcorner P \lrcorner$.

Syntax

After this informal introduction, now we give the full syntax of cJoin with x, y ranging over an infinite set of names ($\vec{y}$ representing a tuple of names); J, K ranging over joining messages; D, E ranging over reaction definitions; M, N ranging over processes without definitions; P, Q ranging over processes; m ranging over molecules; and S ranging over solutions:

$$\begin{array}{rcl} M, N & ::= & 0 \mid x\langle\vec{y}\rangle \mid M|N \\ J, K & ::= & x\langle\vec{y}\rangle \mid J|K \\ D, E & ::= & J \rhd P \mid J \blacktriangleright P \mid D \wedge E \\ P, Q & ::= & M \mid \mathit{abort} \mid \mathbf{def}\ D\ \mathbf{in}\ P \mid [P : Q] \mid P|Q \\ m & ::= & P \mid D \mid \llcorner P \lrcorner \mid \{\!|S|\!\} \\ S & ::= & m \mid m, S \end{array}$$

A process without definitions can be the inert process 0; an emission of a message $\vec{y}$ over port x, written as $x\langle\vec{y}\rangle$; or a parallel composition. A joining message is either an emission of a message $x\langle\vec{y}\rangle$ or a parallel composition of such emissions. A reaction definition is either an association $J \rhd P$ of a joining message J with process P; a merge $J \blacktriangleright P$ with joining message J and process P; or a conjunction of reaction definitions. A process is either a process without definitions; an abort; a definition **def** D **in** P of a reaction definition D over a process P; a negotiation $[P : Q]$, starting as P but continuing as Q if P aborts; or a parallel composition of processes. A molecule is either a process; a reaction definition; a frozen compensation $\llcorner P \lrcorner$ with behaviour P; or a sub-solution $\{\!|S|\!\}$. Finally, a solution is a molecule or a comma separated collection of molecules.

Semantics

Next, we give a number of semantic rules with their explanation below:

$$
\begin{array}{lll}
\textsc{Str-def} & \textbf{def } D \textbf{ in } P & \rightleftharpoons \\
 & & D\sigma_{dn(D)}, P\sigma_{dn(D)} (\text{range } (\sigma_{dn(D)}) \text{ globally fresh}) \\
\textsc{Red} & J \rhd P, J\sigma & \rightarrow \quad J \rhd P, P\sigma \\
\textsc{Str-cont} & [P : Q] & \rightleftharpoons \quad \{\!|P, \llcorner Q \lrcorner|\!\} \\
\textsc{Abort} & \{\!|abort \mid P, \llcorner Q \lrcorner|\!\} & \rightarrow \quad Q \\
\textsc{Commit} & \{\!|M \mid \textbf{ def } D \textbf{ in } 0, \llcorner Q \lrcorner|\!\} & \rightarrow \quad M \\
\textsc{Merge} & J_1 \mid \ldots \mid J_n \blacktriangleright P, \bigotimes_i \{\!|J_i\sigma, S_i, \llcorner Q_i \lrcorner|\!\} & \rightarrow \\
 & & J_1 \mid \ldots \mid J_n \blacktriangleright P, \{\!|\bigotimes_i S_i, P\sigma, \llcorner Q_1 \mid \ldots \mid Q_n \lrcorner|\!\}
\end{array}
$$

The rule Str-def is responsible for opening reaction definitions and generating globally fresh names for the defined names (dn returns the defined names within a reaction definition) in D and P, ensuring that the names remain restricted to D and P. σ represents the substitution of names $x_1, \ldots, x_n$ by names $y_1, \ldots, y_n$ such that $dom(\sigma)$ returns $\{x_1, \ldots, x_n\}$, $range(\sigma)$ returns $\{y_1, \ldots, y_n\}$, and σ_N refers to σ such that $dom(\sigma) = N$. Next, rule Red describes an instance of J being consumed by a reaction rule $J \rhd P$. The consumption is accompanied by substitution of names such that $dom(\sigma)$ is equal to the received names of J. Rule Str-cont shows how a negotiation process corresponds to a solution with two molecules: the process P and the compensation Q which is initially a frozen process waiting to be activated. The rule Abort is the activation of the compensation Q caused by an *abort* process. On the other hand, Commit triggers when the negotiation terminates successfully. Note that at this point the compensation is discarded. Finally, the Merge rule merges several negotiations into one, enabling interaction among negotiations. Note that $\bigotimes_i m_i$ is a shorthand for $m_1, \ldots, m_n$ where m is a molecule.

Example

The example in cJoin is given below starting with the basic activities (In the example below we ignore name locality and assume the names are global.):

$$
\begin{array}{lll}
\mathit{Credit} & \stackrel{\text{def}}{=} & \textbf{def } \mathit{Payment}\langle\rangle \rhd [\\
 & & \quad \textbf{def } \mathit{pay}\langle\$\rangle \rhd \mathit{paymentOK}\langle\rangle \\
 & & \quad\quad \wedge \mathit{pay}\langle\$\rangle \rhd \mathit{abort}\langle\rangle \\
 & & \quad \textbf{in } \mathit{credit}\langle\rangle : \mathit{abortAll}\langle\rangle] \textbf{ in } \mathit{Payment}\langle\rangle \\
\mathit{Refund} & \stackrel{\text{def}}{=} & \textbf{def } \mathit{paymentOK}\langle\rangle \rhd \mathit{refunded}\langle\rangle \\
 & & \quad \wedge \mathit{paymentOK}\langle\rangle \rhd \mathit{Operator}\langle\rangle \\
 & & \textbf{in } \mathit{refundRequest}\langle\rangle
\end{array}
$$

In this example, we start by considering the *Credit* activity since the *Order* activity in cJoin is peculiar. Note that the possibility of failure is modelled through the spec-

ification of two entries for $pay\langle\$\rangle$. If an activity succeeds, $paymentOK\langle\rangle$ is used to communicate the success to the main transaction. Otherwise, in case an abort occurs, $abortAll\langle\rangle$ communicates the failure to the main transactions, causing the whole transaction to start compensation. A similar approach is used to model the failure of *Refund*, triggering the *Operator* activity in case of failure. Note that the execution of *Refund* depends on *paymentOK*. This ensures that refund is only given if the payment has been successful. The same approach is used for the other compensating acitivities.

Next, we consider the encoding of the offers:

$$
\begin{array}{lcl}
\mathit{Offer}_1 & \stackrel{\text{def}}{=} & \textbf{def}\ \mathit{proposeOffer}_1\langle\rangle \rhd [\\
& & \quad \textbf{def}\ \mathit{proposeOffer}_1\langle\$\rangle \rhd \mathit{offerOK}\langle\rangle \\
& & \quad\quad \wedge\ \mathit{proposeOffer}_1\langle\$\rangle \rhd \mathit{abort}\langle\rangle \\
& & \quad \textbf{in}\ \mathit{offerReq}_1\langle\rangle \\
& & \quad : \textbf{def}\ \mathit{offerReq}_2\langle\rangle \blacktriangleright \mathit{proposeOffer}_2\langle\$\rangle \textbf{in}\ \mathit{Offer}_2] \\
& & \textbf{in}\ \mathit{proposeOffer}_1\langle\rangle \\
\mathit{Offer}_2 & \stackrel{\text{def}}{=} & \textbf{def}\ \mathit{proposeOffer}_2\langle\rangle \rhd [\\
& & \quad \textbf{def}\ \mathit{proposeOffer}_2\langle\$\rangle \rhd \mathit{offerOK}\langle\rangle \\
& & \quad\quad \wedge\ \mathit{proposeOffer}_2\langle\$\rangle \rhd \mathit{abort}\langle\rangle \\
& & \quad \textbf{in}\ \mathit{offerReq}_2\langle\rangle : 0]\ \textbf{in}\ \mathit{Offer}_2\langle\rangle
\end{array}
$$

Note that if Offer_1 fails, Offer_2 is triggered, while if the latter fails, the exception handler is the inert process, i.e. failure does not disrupt the overall transaction. However, if either succeeds, *offerOK* communicates the success to the main transaction.

What follows is the definition of the courier booking:

$$
\begin{array}{lcl}
\mathit{Couriers} & \stackrel{\text{def}}{=} & \textbf{def}\ \mathit{CourierBooking}\langle\rangle \rhd [\\
& & \quad \textbf{def}\ \mathit{courierOK}_1\langle\rangle \mid \mathit{courierOK}_2\langle\rangle \\
& & \qquad\qquad \blacktriangleright \mathit{Cancel}_2\langle\rangle \mid \mathit{CourierOK}\langle\rangle \\
& & \qquad \wedge\ \mathit{courierOK}_1\langle\rangle \mid \mathit{courierFailed}_2\langle\rangle \blacktriangleright \mathit{CourierOK}\langle\rangle \\
& & \qquad \wedge\ \mathit{courierFailed}_1\langle\rangle \mid \mathit{courierOK}_2\langle\rangle \blacktriangleright \mathit{CourierOK}\langle\rangle \\
& & \qquad \wedge\ \mathit{courierFailed}_1\langle\rangle \mid \mathit{courierFailed}_2\langle\rangle \blacktriangleright \mathit{abortAll}\langle\rangle \\
& & \quad \textbf{in}\ \mathit{Courier}_1 \mid \mathit{Courier}_2 \\
& & \quad : \mathit{abortAll}\langle\rangle]\ \textbf{in}\ \mathit{CourierBooking}\langle\rangle \\
\mathit{Courier}_1 & \stackrel{\text{def}}{=} & \textbf{def}\ \mathit{bookingReq}_1\langle\rangle \rhd \mathit{courierOK}_1\langle\rangle \\
& & \quad \wedge\ \mathit{bookingReq}_1\langle\rangle \rhd \mathit{courier1Failed}\langle\rangle \\
& & \textbf{in}\ \mathit{BookCourier}_1\langle\rangle \\
\mathit{Courier}_2 & \stackrel{\text{def}}{=} & \textbf{def}\ \mathit{bookingReq}_2\langle\rangle \rhd \mathit{courierOK}_2\langle\rangle \\
& & \quad \wedge\ \mathit{bookingReq}_2\langle\rangle \rhd \mathit{courierFailed}_2\langle\rangle \\
& & \textbf{in}\ \mathit{BookCourier}_2\langle\rangle
\end{array}
$$

The courier booking initiates the bookings in parallel $\mathit{Courier}_1 \mid \mathit{Courier}_2$ and depending on the outcome, chooses a course of action. The first case considered is when both succeed, where the second courier is cancelled through Cancel_2 while *CourierOK* is signalled to the main transaction. Similarly, if either succeeds, success is reported to the main transaction, while *abortAll* is signalled if both fail.

Finally, we define the interaction with the client and the overall transaction:

$$
\begin{array}{lll}
\mathit{Client} & \stackrel{\text{def}}{=} & \textbf{def}\ \mathit{OrderItems} \rhd [\\
& & \quad \textbf{def}\ \mathit{shopResp}\langle \mathit{ok} \rangle \rhd \mathit{itemsBought}\langle\rangle \\
& & \quad\quad \wedge\ \mathit{shopResp}\langle \mathit{failed} \rangle \rhd \mathit{abort} \\
& & \quad \textbf{in}\ \mathit{order}\langle \mathit{info} \rangle : 0] \ \textbf{in}\ \mathit{OrderItems}\langle\rangle \\
\mathit{Order} & \stackrel{\text{def}}{=} & \textbf{def}\ \mathit{ProcessOrders} \rhd [\\
& & \quad \textbf{def}\ \mathit{info}\langle \mathit{req}, \mathit{conf} \rangle \rhd \mathit{price}\langle \$ \rangle \mid \mathit{items}\langle\rangle \\
& & \quad\quad \wedge\ \mathit{price}\langle \$ \rangle \mid \mathit{credit}\langle\rangle \blacktriangleright \mathit{pay}\langle \$ \rangle \\
& & \quad\quad \wedge\ \mathit{items}\langle\rangle \mid \mathit{pack}\langle\rangle \blacktriangleright \mathit{packReq}\langle \mathit{items} \rangle \\
& & \quad\quad \wedge\ \mathit{offerReq}\langle\rangle \blacktriangleright \mathit{Offers}_1\langle \$ \rangle \\
& & \quad\quad \wedge\ \mathit{abortAll} \rhd \mathit{abort} \\
& & \quad\quad \wedge\ \mathit{creditOK} \mid \mathit{packingOK} \mid \mathit{courierOK} \rhd \mathit{shopResp}\langle \mathit{ok} \rangle \\
& & \quad \textbf{in}\ \mathit{sell}\langle \mathit{req}, \mathit{conf} \rangle \\
& & \quad : \mathit{shopResp}\langle \mathit{failed} \rangle \mid \mathit{Withdraw} \mid \mathit{Refund} \mid \mathit{Unpack} \mid \\
& & \quad\quad \mathit{Cancel}_1 \mid \mathit{Cancel}_1] \\
& & \textbf{in}\ \mathit{ProcessOrders}\langle\rangle \\
\mathit{Transaction} & \stackrel{\text{def}}{=} & \textbf{def}\ \mathit{order}\langle \mathit{info} \rangle \mid \mathit{sell}\langle \mathit{req}, \mathit{conf} \rangle \blacktriangleright \mathit{info}\langle \mathit{req}, \mathit{conf} \rangle \\
& & \textbf{in}\ \mathit{Client} \mid \mathit{Order} \mid \mathit{Offer}_1 \mid \mathit{Credit} \\
& & \mid \mathit{Pack} \mid \mathit{Couriers}
\end{array}
$$

Note that the processing of a client order starts at *Transaction* which glues together all the processes by starting them in parallel (*Order*, *Client*, Offer_1, etc) and merging (through the merge ($\blacktriangleright$) operator) negotiations defined in the sub-processes, allowing interactions to take place across negotiation boundaries. In this example, *Transaction* merges *Client* and *Order* through the merge on *order* and *sell*. Similarly, *Order* merges itself with *Credit* through the merge on *credit* and so on.

Since in cJoin there is no explicit construct for compensation, the approach taken is that when a process fails, it sends a signal over *abortAll* which is received by the *Order* negotiation. This, in turn, compensates for all the activities which have been completed. In order to detect whether an activity needs compensation or not, the signals *paymentOK*, *packingOK*, etc are used since these indicate the successful completion of the respective activities.

4.7 Comparison of Choreography Approaches

A major dividing line among the choreography approaches is whether static or dynamic compensation is supported[5]: $\pi\mathtt{t}$ calculus, webπ and cJoin only support static compensation. There are other distinctive features which isolate particular notations from the rest. For example, SOCK is the only notation allowing total freedom as to the order in which compensations are installed. On the other hand, dcπ is the only notation which associates compensation installation with inbound interaction. webπ is the only notation which ignores transaction nesting and treats all transactions as parallel transactions. A direct consequence is that in webπ the failure of a transaction does not affect any other transaction unlike the case of the other notations.

Other features which are common to all notations will be discussed in Section 6 where orchestration approaches are compared to choreography approaches.

[5]Dynamic compensation is when compensation is constructed dynamically during execution while static compensation is predefined.

5 Business Process Execution Language for Web Services

With the advancement of the Service Oriented Architecture (SOA), promoting software integration across organisational boundaries, various challenges arose — particularly the challenge of programming complex services and interactions. Among the suggested languages to tackle such a problem were Microsoft's XLANG [Tha01] and IBM's WSFL [Ley01]. Business Process Execution Language for Web Services, known as BPEL4WS [bpe03] and more recently WS-BPEL [bpe07] (or BPEL for short) is the offspring of both XLANG and WSFL, integrating XLANG's structured nature with WSFL graph-based approach. BPEL's semantics is given in textual descriptions, giving rise to a lot of work which strives to provide a formal semantics for BPEL — motivated by the need of clearer semantics and verification of various properties.

It is beyond the scope of the review to give a detailed overview of BPEL (for a more detailed overview we refer the read to for example [LM07]). Rather, we focus on the compensation mechanism of BPEL, giving a simple example for illustration purposes. Next, we go through a number of formalisms which have been suggested to provide a formal semantics for BPEL (distinguishing between those which include BPEL's compensation construct from those which do not). We conclude this section by considering some critiques of BPEL.

5.1 Overview

BPEL processes fall under two categories: abstract processes and executable processes. The former's aim is to give a high-level description of the interactions that a particular party may have in a conversation. Such processes cannot be executed because they lack details. On the other hand, executable processes are processes describing business logic in full detail — enabling such processes to be executed. The main aim for providing this duality is two-fold: on the one hand, a business will not need to disclose the internal logic to other businesses, but it suffices to simply disclose abstract processes to enable interaction; and on the other hand, it provides a separation of concerns between the interactions and the business logic — enabling changes in the business logic to leave the interaction unaffected.

In this overview, we focus on executable BPEL processes, particularly on the fault, compensation and termination handling mechanisms. BPEL provides various basic activities and structured activities. Basic activities include the following:

Receive Waits for a request (invocation) from a partner.

Reply Allows a business process to send a message in reply to one received earlier (by a receive activity).

Invoke Invokes a remote operation provided by a partner participating in an interaction. The invoked operation can also send back a response.

Assign Assigns new values to variables.

Validate Validates values of variables according to their definition.

Throw Generates a fault from inside a business process.

Wait Waits for a given period of time.

Empty Does nothing but is considered as a successful activity.

Exit Immediately ends the enclosing business process.

Rethrow Rethrows a fault to the next outer scope.

Compensate Activates the installed compensation.

ExtensionActivity Can be used to extend BPEL by introducing a new type of activity.

Structured activities include the following:

Sequence Is used to define a collection of activities to be performed in sequential order.

If Selects exactly one activity for execution from a set of choices.

While Defines a child activity to be repeated as long as the specified condition is true.

RepeatUntil Repeats a child activity until a specified condition becomes true.

ForEach Repeats a child activity for a specified number of times.

Pick Waits for one of several possible events to occur (either a message arrival or a time to elapse) and triggers the associated child activity.

Flow Specifies one or more activities to be executed concurrently.

Scope Is used to define a nested activity with associated partner links, message exchanges, variables, correlation sets, fault handlers, compensation handler, termination handler, and event handlers. (See Section 5.1.1 for more details.)

CompensateScope Acts as an enclosing scope for the execution of a compensation.

After the above short introduction to the activities available in BPEL, we proceed to tackle BPEL's compensation mechanism in detail. Compensation in BPEL has a close relationship to scopes, faults and termination handling. In the following subsections, we will thus consider these mechanisms in detail.

5.1.1 Scopes

A scope in BPEL is the mechanism which associates compensation to a unit of execution. Furthermore, each scope also provides event handlers, fault handlers and a termination handler to the scope-enclosed activity.

The life-cycle of a scope can be summarised as follows (more details in the subsequent): (i) the enclosed activity starts executing; (ii) if a fault occurs, then this is thrown for the fault handler to handle it; (iii) if a forced termination occurs, the executing activity is terminated, executing the termination handler; (iv) if the scope completes successfully, the corresponding compensation is installed; (v) if a fault, termination, or another compensation handler calls for the scope's compensation to be executed, then the installed compensation is activated.

5.1.2 Fault Handling

A scope may have a number of *catch* statements which intercept a fault. If such a statement does not exist for a particular kind of fault, then the fault is rethrown to the next higher level. Upon the activation of a fault handler, the active activities (including event handlers) within the enclosing scope are automatically terminated (although some kind of activities may be allowed to continue till completion). If not specified, by default the fault handler activates the installed compensations of any previously completed scopes (within the fault handler's scope) and rethrows the fault to the next higher level. If a fault occurs during fault handling this is thrown to the next higher scope.

5.1.3 Compensation Handling

Although there is an explicit activity which is responsible for activating installed compensations, this cannot be invoked from any part of a scope but only from within fault handlers, termination handlers or other compensation handlers. Thus, the use of compensation is limited to be used as a part of the fault-handling mechanism. Therefore compensations can be triggered in three scenarios: (i) either a fault has been thrown and any installed compensations are executed; (ii) the compensation handler triggers the compensation for previously completed nested scopes; (iii) or the terminations handler triggers the compensation in order to compensate for a forcedly terminated scope.

Scopes within compensation handlers

A compensation scope may have other scopes within it and such scopes may have associated compensations. However, the outermost scope (of a compensation) may not have compensations as this would be unreachable. In other words, a compensation handler may utilise the compensation mechanism so that either the compensation succeeds or none of it, yet once the compensation completes there is no way of undoing it. In fact, upon the completion of a compensation, any installed compensations are discarded.

Default compensation handler

If left unspecified, the default compensation handler of a scope simply activates the compensations installed for the next-lower level scopes. Note that a scope can only trigger the compensation of its immediate child-scopes (which in turn can trigger that of their children, etc).

Default compensation order

Although a compensation can be application specific, if left unspecified, a default compensation order is utilised. Compensation ordering is a complex issue which different compensation mechanisms handle differently. BPEL specifies that any order dependencies which were in place during normal execution, must also be in place (in reverse) during compensation. Such order dependencies can either be the result of the sequence structured activity or due to explicit link dependencies. Link dependencies are tricky because links may be present across scope boundaries. However, as long as there are no cyclic dependencies, the activity ordering issue can be resolved. Otherwise, a BPEL process with cyclic dependencies is not a valid BPEL process.

Process state usage

Since activities in BPEL may depend on data, it is crucial to determine the data

values which are used during compensation. To this end *scope snapshots* are taken at the point of a scope completion so that when the corresponding compensation is executed, it has access to the data values as were preserved at the end of that scope execution. A compensation handler also has access to the state of the enclosing scope (from where the compensation was activated).

5.1.4 Termination Handling

Upon the throw of a fault, any remaining activities executing within the scope are terminated (see Section 5.1.2). Once forced termination is completed, the termination handler is executed. The aim of the termination handler is to execute any necessary actions in order to correctly terminate the scope. If left unspecified, the termination handler automatically invokes the compensation handler in order to compensate for any successfully completed activities. Note that if an unhandled fault occurs during termination handling, this is not rethrown because the enclosing scope has already faulted.

5.2 Example

In the example below, we consider a very simple scenario where a bookstore first updates the stock levels — checking that there is enough stock, and subsequently packs the books and charges the customer's bank account concurrently. Note that the implementation details are left out.

```
<process name="bookSellingTransaction">

  <documentation xml:lang="EN">
    A WS-BPEL process for handling a purchase order.
  </documentation>

    ... partnerLinks, variables, etc

  <faultHandlers>
    <catchAll>
        <compensate />
    </catchAll>
  </faultHandlers>

  <sequence>

    <scope name="stockUpdate">
```

```
  <documentation>
    This is the scope for handling the stock level.
  </documentation>

  <faultHandlers>
<!-- This is the default fault handler>
    <catchAll>
      <sequence>
        <compensate />
        <rethrow />
      </sequence>
    </catchAll>
  </faultHandlers>

  <compensationHandler>
     ... reupdate stock levels
  </compensationHandler>

  ... checkstock and decrease
</scope>

<flow>
  <scope name="packOrder">

    <documentation>
      This is the scope for packing the order.
    </documentation>

    <compensationHandler>
       ... unpack
    </compensationHandler>

  <!-- This is the default termination handler>
    <terminationHandler>
       <compensate />
    </terminationHandler>

    ...packing
  </scope>

  <scope name="chargeCustomer">

    <documentation>
      This is the scope for charging the customer's account.
    </documentation>

    <compensationHandler>
       ... refund customer
```

```
        </compensationHandler>

          ... charge customer's bank account
      </scope>

    </flow>
  </sequence>
</scope>
```

The encoding in BPEL is done using three scopes — one for each activity with a compensation. This is due to the fact that compensation can only be associated to an activity through a scope. Due to the sequence and concurrency requirement of the example, the corresponding constructs are used to encapsulate the activities accordingly.

Note that in all scopes, the default fault handling mechanism is utilised (only explicitly shown for the scope *stockUpdate*; it is implicit in the other cases). This simply executes the installed compensations (using `<compensate />`) and rethrows the fault to the next higher level scope. Similarly, we only explicitly declare the default termination handler in scope *packOrder* so that, if for example the bank account charge fails while the packing is still executing, compensation is executed (assuming packing has a further nested scope with compensation).

The rest of the details are left out, indicated in the example by `... activity`. These can be implemented using BPEL processes such as `assign` to update stock values, and `invoke` and `receive` to communicate with the bank and the warehouse (where the packing takes place).

Figure 4 shows a graphic representation of the example just described, highlighting the scopes, compensation, fault and termination handlers. The notation used is inspired by the diagrams in the BPEL documentation [bpe07].

5.3 BPEL Formalisations

A lot of work has been done to provide a formalisation of BPEL. However, various formalisms proposed for modelling BPEL [HB03, AFFK04, KvB04, HMMR04, FBS04, PRB04, WFN04, Vir04, VvdA05, PTBM05, BBM+05, KP05, YTX+06, PZWQ06, Nak06, WDW07, MR08, yLsL09] do not take into consideration the compensation mechanism. The reason is that, given the complexity of BPEL's compensation mechanism, works which do not focus on the error-handling aspect are likely to leave compensations out. Since our review is focused on compensation semantics, we do not tackle such formalisms. However, some formalisms discussed below (which include compensation) are based on some of these referenced works.

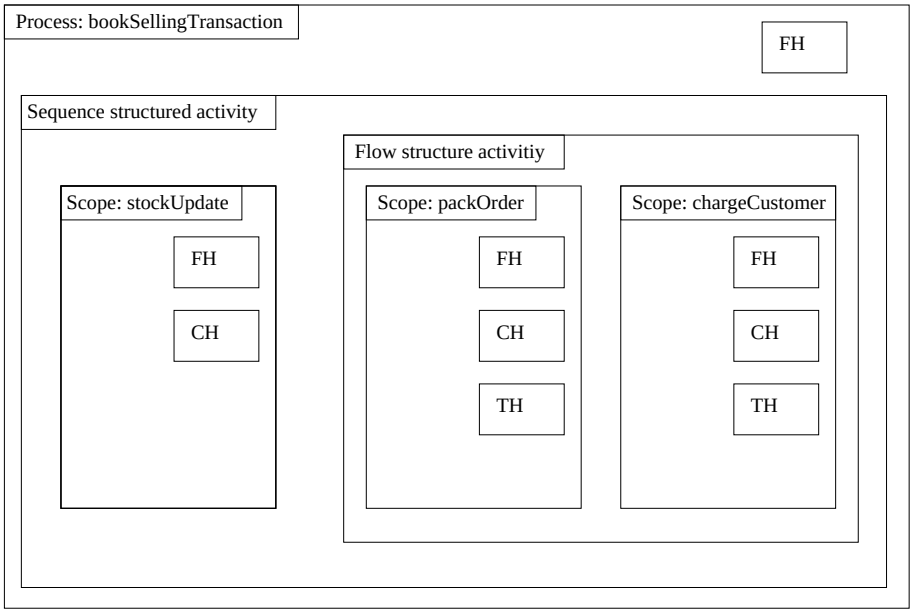

Figure 4: A representation of the BPEL example.

Two formalisms which have already been tackled in this report have also been proposed as formalisms for modelling BPEL. These are StAC (see Section 3.3) — mapping from BPEL to StAC in [BFN05] and $\texttt{web}\pi_\infty$ (see Section 4.3) — mapping from BPEL to $\texttt{web}\pi_\infty$ in [LM07]. In particular, StAC has been combined with the B notation [Abr96] (enabling the representation of state operations), to model a substantial subset of BPEL, including compensations. The translation from BPEL to $\texttt{web}\pi_\infty$ is particularly interesting since $\texttt{web}\pi_\infty$ only provides one error handling mechanism (unlike BPEL). The translation does not handle the whole of BPEL but it focuses on event, fault and compensation handlers, all of which are modelled through the single mechanism provided by $\texttt{web}\pi_\infty$.

In the rest of this section, we consider a number of other BPEL formalisations, focusing on the compensation mechanism. For a complete overview of BPEL formalisms we refer the reader to [vBK05] (although a number of formalisms have emerged since). We group BPEL formalisms according to their type, and after giving a short overview, we comment about which parts of BPEL are handled by the formalisations.

5.3.1 Formalisms based on Abstract State Machines

An abstract state machine (ASM) involves a state made up of arbitrary variables and a number of transition rules. Each rule consists of a boolean condition and a number

of assignment statements which can be applied on the state. The boolean condition is used to decide whether the transition is applicable or not at a particular state. The ASM changes state by applying the assignments of the enabled transitions on the state.

ASM-based BPEL models

ASMs have been used to model BPEL by Farahbod et al. [Far04, FGV05] and by Fahland and Reisig [FR05, Fah05]. It is beyond the scope of this short overview to give the full details of each of these formalisms. However, we shall give some intuition about the method used by Fahland and Reisig as an example. Consider the following activity definition:

$$\textsc{ExecuteScope}_{compensate,\ pattern}(\text{sI} \in \mathit{SubInstance}, \text{scope} \in \mathit{Scope}) \equiv$$

if $\mathit{receivedCompensationCalls}(\text{sI}, \text{scope}) \neq \emptyset$
then $\textsc{HandleCompensationCall}_{pattern}(\text{sI}, \text{scope})$

onSignal *signalCompleted* **from** *scopeCompensationHandler(scope)*
in sI **via** $\mathit{signalChannel}_{up}$ **do** $\textsc{ConfirmComepsantion}(\text{sI}, \text{scope})$

if $\mathit{faultChannel}_{up}(\text{sI}, \mathit{scopeCompensationHandler}(\text{scope}), \text{scope}) \neq \emptyset$
then $\textsc{PropagateFaultsFromCH}(\text{sI}, \text{scope})$

The above rule handles a scope after completion, i.e. waiting for the possibility of executing its compensation. If a call for the execution of the compensation is received ($\mathit{receivedCompensationCalls}(\text{sI}, \text{scope}) \neq \emptyset$), the compensation handler is called. Subsequently, if the compensation completes, *signalCompleted* is received and a confirmation is sent to the originator of the compensation. Otherwise, if the compensation fails, a signal is received on *faultChannel* which is propagated so that the appropriate scope can handle it. Note that parametrisation is crucial to determine the correct instance and scope in which the activity is operating.

Similar to this approach, a rule is available for every BPEL construct. Therefore, in order to model a BPEL process, one simply has to translate the process into the initial state of the ASM and then simply apply the rules to model the execution of the process.

In what follows, we comment on both works with regards to their treatment of BPEL.

Comments

Farahbod et al. give a mapping from each BPEL construct (including fault, com-

pensation, and termination constructs) to an ASM agent thus preserving a direct relationship to BPEL. Fahland and Reisig consider their work to be very similar to that of Farahbod et al. and their work also handles all of BPEL.

5.3.2 Formalisms based on Petri Nets

A Petri net is a directed, connected, and bipartite graph in which each node is either a place or a transition. Note that places cannot be directly connected to other places and neither can transitions be directly connected to other transitions. Places may be occupied by tokens. For a transition to be enabled, each place connected by an incoming arrow to the transition must contain at least one token. When a transition fires, it consumes a token from each incoming place and adds one in each outgoing place. Note that a transition firing is an atomic operation.

Petri Net-based BPEL models

There are three major works which model BPEL processes in Petri nets:

(i) A Petri net semantics for BPEL 1.1 was developed by Hintz et al. [HSS05] based on previous work by Schmidt and Stahl [SS04]. The work was later enhanced to cover BPEL 2.0 by Lohmann [Loh07];

(ii) Another Petri net-based BPEL semantics was given by Ouyang et al. [OVvdA+07];

(iii) He et al. [HZWL08] use general stochastic high-level Petri nets (GSHLPN) to model BPEL.

The main idea behind modelling BPEL in terms of Petri nets is to define a Petri net pattern for each of BPEL's activities and then simply glue the patterns together to form a complete model of a BPEL process. This approach has the advantage of preserving the structure of BPEL. In what follows, we give an example of a pattern related to the compensation mechanism taken from [OVvdA+07]. Consider Figure 5 where a compensation invocation takes place, attempting to compensate for scope Q_1.

Note that the call to compensate scope Q_1 originates from within a *FH/CH* scope representing a fault handler or a compensation handler. For the invocation to be successful (i.e. causing the compensation activity to actually execute), the scope being compensated should have completed. Upon completion, a scope preserves a scope snapshot so that the ending state is available for the compensation activity to use during its execution. Note how, in the absence of a snapshot, the compensation

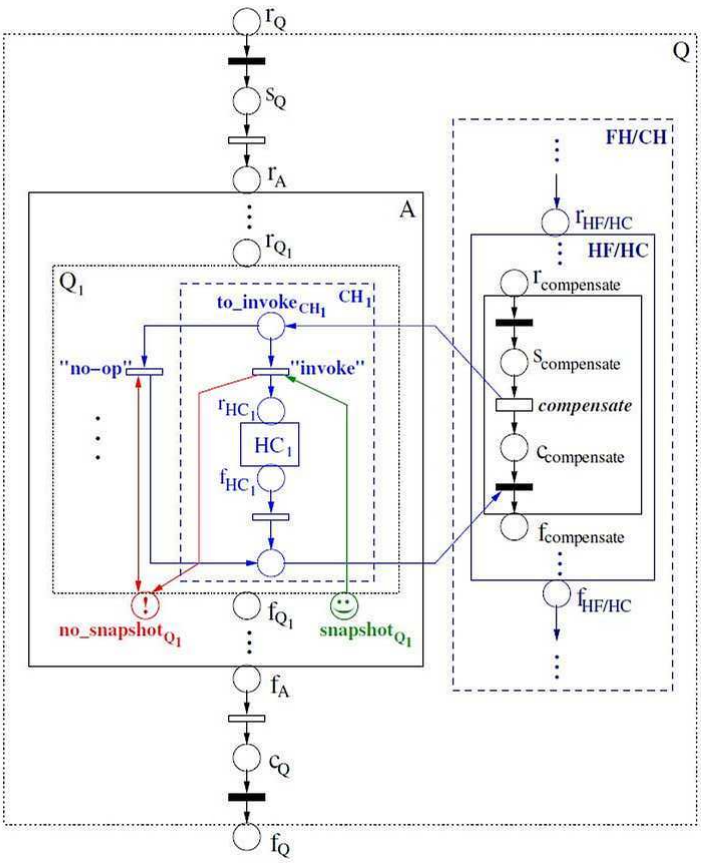

Figure 5: A Petri net pattern for handling a compensation invocation (taken from [OVvdA+07]).

activity is by-passed and behaves as no-operator. Otherwise, the invocation of the compensation handler takes place. In any case, after the completion of the invocation, control is passed back to the caller *FH/CH*.

Comments

The work by Hintz et al. [HSS05] and Lohmann [Loh07] have already been compared by Lohmann et al. [LVOS09] with the work by Ouyang et al. [OVvdA+07]. They are both feature-complete (i.e. model all features of BPEL) but differ in some aspects. Of particular interest is that the work of Ouyang et al., activities stop immediately in case of a fault, while in the other case activities may take time to stop (although the fault handler does not start unless the activities have terminated). Finally, the work by He et al. [HZWL08] mainly focuses on the need to model the time aspect of transactions, enabling performance evaluation. Furthermore, they exploit GSHLPN's

two-level priority transactions to model interrupt behaviour, facilitating the encoding of exception, fault and compensation-related events.

5.3.3 Formalisms based on Process Algebras

A number of process algebras which handle compensations have already been introduced in Section 4. In this section, we aim at considering process algebras which have been used for modelling BPEL.

Process algebra-based BPEL models

Process algebras are by far the most commonly used kind of formalisms to model BPEL. Here we list six works in this respect:

(i) A two-way mapping between LOTOS and BPEL [Fer04b, Fer04a];

(ii) *BPEL0*, a calculus by Pu et al. [PZQ+06];

(iii) A mapping from BPEL to finite state process notation (FSP) [FUMK06, Fos06];

(iv) BPEL_{fct}, a calculus focusing on BPEL's fault, compensation and termination (fct) handling by Eisentraut and Spieler [ES08];

(v) BP-calculus, a calculus by Abouzaid and Mullins [AM08];

(vi) B*lite*, a language meant as a simplification of BPEL [LPT08b].

Comments

The work by Ferrara [Fer04b, Fer04a] is notorious for its two-way mapping between LOTOS and BPEL, allowing the programmer freedom of choice while enjoying the benefits of both languages. The mapping includes compensation but does not consider all the aspects of BPEL.

BPEL0 [PZQ+06] is a language aimed at studying scope-based compensation languages and thus models only the key features of BPEL. In particular, *BPEL0* focuses on *compensation contexts* and *compensation closures*. Informally, a compensation context is a compensation container which is currently being accumulated with new compensations, while a compensation closure is a compensation container which is awaiting to be activated but no more compensations can be accumulated.

Foster et al. [FUMK06, Fos06] present a mapping from BPEL to finite state process notation (FSP) in order to verify a design in message sequence charts (which is also

translated in FSP) against its implementation. The mapping from BPEL to FSP includes compensation but is not feature complete.

Eisentraut and Spieler particularly used process algebra to study BPEL's fct mechanism [QWPZ05, ES08][6]. The aim of the work is not to cover all of BPEL's semantics but rather to capture in full detail the BPEL's fct mechanism. Two important aspects covered are (i) *all-or-nothing semantics* — the fact that compensations can be compensated, thus semantically undoing all the effects of a failed transaction; and (ii) *coordinated interruption* — forcing parallel siblings of a failed activity to terminate as soon as possible and simultaneously.

Abouzaid and Mullins [AM08] propose BP-calculus for which they give the corresponding semantics through BPEL, i.e. the mapping is from BP-calculus to BPEL and not the other way round as in the case of the other formalisms. In this fashion they allow modelling and verification of workflows in a formal language, which can then be directly implemented in BPEL. Their mapping includes all the constructs of BPEL.

B*lite* [LPT08b] is a formalisation of a significant subset of BPEL (including compensation), aiming to clarify intricate issues in BPEL. A mapping from B*lite* to COWS (see Section 4.5) has also been defined, enabling the application of formal tools available for COWS to be applied to B*lite* and therefore to BPEL.

5.4 Comparison of BPEL Formalisation Approaches

The main motivation behind all BPEL formalisations is to give a precise semantics of BPEL which would enable formal reasoning to be applied. Certain formalisation techniques provide an encoding which is closer to the original BPEL process and thus arguably more understandable. In particular, this is true for ASMs which provide a means of translating a BPEL process into a mathematical object clearly showing how the state of processes evolves. Other formalisations are mainly attractive due to their support for particular proof techniques. For example bisimulation techniques for process algebras are highly developed and thus notions of equivalence among BPEL processes (encoded in process algebras) can be defined more straightforwardly. While also providing a range of formal techniques, Petri nets have the advantage of preserving the structure of BPEL processes and clearly showing the flow of control among such processes.

[6] Although the first work referenced does not take termination handling into consideration.

6 Comparison of Orchestration and Choreography Approaches

In this section we aim to consider the commonalities of both orchestration approaches and choreography approaches and thus compare the overall approaches of orchestration versus choreography.

The most basic difference between orchestration and choreography approaches is that the latter requires the processes to communicate together while the former assumes a central coordinator and thus processes need not be aware of each other. The consequence is that in the case of choreography approaches various composition operators can be encoded in terms of the communication mechanism provided by the approach. On the other hand, orchestration approaches have to provide all the operators explicitly. This explains why in general far less operators are provided in the choreography approaches. As an extreme consider `web`π which only provides the transaction operator over and above the communication operators. Other choreography approaches such as COWS provide a very basic set of operators but define higher level ones in terms of the basic.

Another consequence of the fact that choreography approaches include communication among processes is that these show interaction among different parties explicitly. For this reason a number of choreography approaches also support the notion of a location where a process is executing. On the other hand, orchestration approaches do not need communication among the participating parties. Due to this and the fact that this necessitates more explicit operators, the composition of activities to form a higher level activity is usually more immediately clear when analysing an orchestrated activity.

7 Conclusions

Fuelled by the streamlining of more complex business logic, the increasing frequency of financial operations, and the increasing degree of cross-entity interaction, modern day transactions require a robust framework, able to deal efficiently with failures. Compensation is a mechanism which has been added over and above traditional transaction mechanisms, enabling them to handle more complex transactions, involving various lower-level transactions. The vast literature on the subject corroborates the relevance of compensations in dealing with complex transactions.

The initial ideas of compensations, have evolved and been integrated with more complex models involving amongst other aspects parallelism, exception handling, transaction nesting and process interaction. Various approaches to several design issues emerge when coming up with a compensation model. This variety is apparent in the differences which exist among different formalisms and models which handle compensations. In order to highlight these differences, we have given an overview of various notations, focusing particularly on their compensation mechanism. Furthermore, through a non-trivial example, we have shown how each of these notations can be used to encode various aspects of compensable transactions, including alternative forwarding and speculative choice. Although, there are other notations catering for compensations, we have included those which propose a substantially different compensation mechanism from other approaches. For example, Acu and Reisig [AR06] extended the work on workflow nets by v.d. Aalst [vdA98] to include compensations. However, the concept of compensation employed is very much like that of Sagas [BMM05]. Similarly, the concepts presented in [GYDuR06] (which we leave out) are almost identical to those in [BLZ03].

The ever increasing interaction among computer systems suggests that the usefulness of compensating transactions will not diminish in the near future. Considerable research has been carried out in the research of the area, particularly by suggesting different formal models of compensation and defining formal semantics for BPEL. However, although BPEL is a *de facto* standard in industrial settings, as yet, there seems to be no accepted standard formalisation of compensating transactions. We hope that this report, reviewing and comparing various works in compensating transactions, contributes towards a more universally accepted view of compensations.

Acknowledgements

We wish to thank Adrian Francalanza for his feedback on preliminary versions of this report.

References

[Abr96] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[AFFK04] Jesús Arias-Fisteus, Luis Sánchez Fernández, and Carlos Delgado Kloos. Formal verification of bpel4ws business collaborations. In *E-Commerce and Web Technologies (EC-Web)*, volume 3182 of *Lecture Notes in Computer Science*, pages 76–85, 2004.

[AKY99] Rajeev Alur, Sampath Kannan, and Mihalis Yannakakis. Communicating hierarchical state machines. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming*, ICAL '99, pages 169–178, London, UK, 1999. Springer-Verlag.

[AM08] Faisal Abouzaid and John Mullins. A calculus for generation, verification and refinement of BPEL specifications. *Electr. Notes Theor. Comput. Sci.*, 200(3):43–65, 2008.

[AR06] Baver Acu and Wolfgang Reisig. Compensation in workflow nets. In *Petri Nets and Other Models of Concurrency (ICATPN)*, volume 4024 of *Lecture Notes in Computer Science*, pages 65–83, 2006.

[BBM+05] Matteo Baldoni, Cristina Baroglio, Alberto Martelli, Viviana Patti, and Claudio Schifanella. Verifying the conformance of web services to global interaction protocols: A first step. In *Formal Techniques for Computer Systems and Business Processes, European Performance Engineering Workshop, EPEW 2005 and International Workshop on Web Services and Formal Methods, WS-FM 2005*, volume 3670 of *Lecture Notes in Computer Science*, pages 257–271, 2005.

[BF04] Michael J. Butler and Carla Ferreira. An operational semantics for stac, a language for modelling long-running business transactions. In *COORDINATION*, volume 2949 of *Lecture Notes in Computer Science*, pages 87–104, 2004.

[BFN05] Michael J. Butler, Carla Ferreira, and Muan Yong Ng. Precise modelling of compensating business transactions and its application to BPEL. *J. UCS*, 11(5):712–743, 2005.

[BHF04] Michael J. Butler, C. A. R. Hoare, and Carla Ferreira. A trace semantics for long-running transactions. In *25 Years Communicating Sequential Processes*, pages 133–150, 2004.

[BLZ03] Laura Bocchi, Cosimo Laneve, and Gianluigi Zavattaro. A calculus for long-running transactions. In *FMOODS*, volume 2884 of *Lecture Notes in Computer Science*, pages 124–138, 2003.

[BMM04] Roberto Bruni, Hernán C. Melgratti, and Ugo Montanari. Nested commits for mobile calculi: Extending join. In *IFIP TCS*, pages 563–576, 2004.

[BMM05] Roberto Bruni, Hernán Melgratti, and Ugo Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 209–220, 2005.

[bpe03] Business process execution language for web services version 1.1, 2003. Available at: http://download.boulder.ibm.com/ibmdl/pub/software/dw /specs/ws-bpel/ws-bpel.pdf (Last accessed: 2010-02-17).

[bpe07] Web services business process execution language version 2.0, 2007. OASIS Standard. Available at: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf (Last accessed: 2010-02-17).

[bpm08] Business process modeling notation, v1.1, 2008. Available at: http://www.bpmn.org/Documents/BPMN_1-1_Specification.pdf (Last accessed: 2010-02-17).

[BR05] Michael J. Butler and Shamim Ripon. Executable semantics for compensating csp. In *EPEW/WS-FM*, pages 243–256, 2005.

[CGV+02] M. Chessell, C. Griffin, D. Vines, M. Butler, C. Ferreira, and P. Henderson. Extending the concept of transaction compensation. *IBM Syst. J.*, 41(4):743–758, 2002.

[DDDGB08] Gero Decker, Remco M. Dijkman, Marlon Dumas, and Luciano García-Bañuelos. Transforming BPMN diagrams into YAWL nets. In *Business Process Management (BPM)*, volume 5240 of *Lecture Notes in Computer Science*, pages 386–389, 2008.

[DDO07] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Formal semantics and analysis of BPMN process models using petri nets, 2007. Preprint (7115).

[DGHW07] Marlon Dumas, Alexander Großkopf, Thomas Hettel, and Moe Thandar Wynn. Semantics of standard process models with OR-joins. In *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS, OTM Confederated International Conferences*, volume 4803 of *Lecture Notes in Computer Science*, pages 41–58, 2007.

[ES08] Christian Eisentraut and David Spieler. Fault, compensation and termination in WS-BPEL 2.0 - a comparative analysis. In *Web Services and Formal Methods (WS-FM)*, volume 5387 of *Lecture Notes in Computer Science*, pages 107–126, 2008.

[Fah05] Dirk Fahland. Complete abstract operational semantics for the web service business process execution language. Informatik-Berichte 190, Humboldt-Universität zu Berlin, 2005.

[Far04] Roozbeh Farahbod. Extending and refining an abstract operational semantics of the web services architecture for the business process execution language. Master's thesis, School of Computing Science, Simon Fraser University, 2004.

[FBS04] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of interacting BPEL web services. In *international conference on World Wide Web, WWW 2004*, pages 621–630, 2004.

[Fer04a] A. Ferrara. Web services: a process algebra approach. Technical report, Università di Roma "La Sapienza", Italy, 2004.

[Fer04b] Andrea Ferrara. Web services: a process algebra approach. In *Service-Oriented Computing - ICSOC 2004*, pages 242–251, 2004.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–385, 1996.

[FGV05] Roozbeh Farahbod, Uwe Glässer, and Mona Vajihollahi. A formal semantics for the business process execution language for web services. In *Web Services and Model-Driven Enterprise Information Services (WS-MDEIS 2005)*, pages 122–133, 2005.

[Fos06] Howard Foster. *A Rigorous Approach to Engineering Web Service Compositions*. PhD thesis, University Of London, 2006.

[FR05] Dirk Fahland and Wolfgang Reisig. ASM-based semantics for BPEL: The negative control flow. In *12th International Workshop on Abstract State Machines, ASM 2005*, pages 131–152, 2005.

[FUMK06] Howard Foster, Sebastián Uchitel, Jeff Magee, and Jeff Kramer. LTSA-WS: a tool for model-based verification of web service compositions and choreography. In *28th International Conference on Software Engineering (ICSE 2006)*, pages 771–774, 2006.

[GLG+06] Claudio Guidi, Roberto Lucchi, Roberto Gorrieri, Nadia Busi, and Gianluigi Zavattaro. SOCK: A calculus for service oriented computing. In *Service-Oriented Computing (ICSOC)*, pages 327–338, 2006.

[GLMZ08] Claudio Guidi, Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. On the interplay between fault handling and request-response service invocations. In *8th International Conference on Application of Concurrency to System Design (ACSD)*, pages 190–198, 2008.

[GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 249–259, 1987.

[GYDuR06] Memon Abdul Ghafoor, Jianwei Yin, Jinxiang Dong, and Maree Mujeeb u Rehman. π_{RBT}-calculus compensation and exception handling protocol. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0:39–47, 2006.

[HB03] Rachid Hamadi and Boualem Benatallah. A petri net-based model for web service composition. In *ADC '03: Proceedings of the 14th Australasian database conference*, pages 191–200, 2003.

[HMMR04] Serge Haddad, Tarek Melliti, Patrice Moreaux, and Sylvain Rampacek. Modelling web services interoperability. In *International Conference on Enterprise Information Systems (ICEIS)*, pages 287–295, 2004.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HSS05] Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming BPEL to petri nets. In *Business Process Management*, volume 3649 of *Lecture Notes in Computer Science*, pages 220–235, 2005.

[HZWL08] Yanxiang He, Liang Zhao, Zhao Wu, and Fei Li. Formal modeling of transaction behavior in WS-BPEL. In *International Conference on Computer Science and Software Engineering, (CSSE)*, pages 490–494, 2008.

[KP05] Raman Kazhamiakin and Marco Pistore. A parametric communication model for the verification of bpel4ws compositions. In *Formal Techniques for Computer Systems and Business Processes, European Performance Engineering Workshop, EPEW 2005 and International Workshop on Web Services and Formal Methods, WS-FM 2005*, volume 3670 of *Lecture Notes in Computer Science*, pages 318–332, 2005.

[KvB04] Mariya Koshkina and Franck van Breugel. Modelling and verifying web service orchestration by means of the concurrency workbench. *ACM SIGSOFT Software Engineering Notes*, 29(5):1–10, 2004.

[Ley01] F. Leymann. WSFL — web services flow language, 2001. IBM Software Group.

[LM07] Roberto Lucchi and Manuel Mazzara. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, 70(1):96–118, January 2007.

[LMSMT06] Ruggero Lanotte, Andrea Maggiolo-Schettini, Paolo Milazzo, and Angelo Troina. Modeling long-running transactions with communicating hierarchical timed automata. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 4037 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2006.

[LMSMT08] Ruggero Lanotte, Andrea Maggiolo-Schettini, Paolo Milazzo, and Angelo Troina. Design and verification of long-running transactions in a timed framework. *Sci. Comput. Program.*, 73:76–94, 2008.

[Loh07] Niels Lohmann. A feature-complete petri net semantics for WS-BPEL 2.0. In *Web Services and Formal Methods (WS-FM)*, volume 4937 of *Lecture Notes in Computer Science*, pages 77–91, 2007.

[LPT07] Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. A calculus for orchestration of web services. In *Programming Languages and Systems (ESOP)*, volume 4421 of *Lecture Notes in Computer Science*, pages 33–47, 2007.

[LPT08a] Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. A calculus for orchestration of web services. Technical report, Dipartimento di Sistemi e Informatica, Univ. Firenze, 2008. http://rap.dsi.unifi.it/cows.

[LPT08b] Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. A formal account of WS-BPEL. In *Coordination Models and Languages (COORDINATION)*, volume 5052 of *Lecture Notes in Computer Science*, pages 199–215, 2008.

[LVOS09] Niels Lohmann, Eric Verbeek, Chun Ouyang, and Christian Stahl. Comparing and evaluating Petri net semantics for BPEL. *International Journal of Business Process Integration and Management (IJBPIM)*, 4(1):60–73, 2009.

[LZ05] Cosimo Laneve and Gianluigi Zavattaro. Foundations of web transactions. In *FoSSaCS*, volume 3441 of *Lecture Notes in Computer Science*, pages 282–298, 2005.

[LZ09] Ivan Lanese and Gianluigi Zavattaro. Programming sagas in SOCK. In *Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009*, pages 189–198, Hanoi, Vietnam, 2009.

[LZH07] Jing Li, Huibiao Zhu, and Jifeng He. Algebraic semantics for compensable transactions. In *Theoretical Aspects of Computing (ICTAC)*, volume 4711 of *Lecture Notes in Computer Science*, pages 306–321, 2007.

[LZPH07] Jing Li, Huibiao Zhu, Geguang Pu, and Jifeng He. Looking into compensable transactions. *Software Engineering Workshop, Annual IEEE/-NASA Goddard*, 0:154–166, 2007.

[MG05] Manuel Mazzara and Sergio Govoni. A case study of web services orchestration. In *Coordination Models and Languages, 7th International Conference, COORDINATION 2005*, volume 3454 of *Lecture Notes in Computer Science*, pages 1–16, 2005.

[ML06] Manuel Mazzara and Ivan Lanese. Towards a unifying theory for web services composition. In *Web Services and Formal Methods, Third International Workshop, WS-FM 2006*, volume 4184 of *Lecture Notes in Computer Science*, pages 257–272, Vienna, Austria, 2006.

[MR08] Radu Mateescu and Sylvain Rampacek. Formal modeling and discrete-time analysis of bpel web services. In *Advances in Enterprise Engineering I (CIAO!) and 4th International Workshop EOMAS*, volume 10 of *Lecture Notes in Business Information Processing*, pages 179–193, 2008.

[Nak06] Shin Nakajima. Model-checking behavioral specification of BPEL applications. *Electr. Notes Theor. Comput. Sci.*, 151(2):89–105, 2006.

[OVvdA+07] Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H. M. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Sci. Comput. Program.*, 67(2-3):162–198, 2007.

[PRB04] Marco Pistore, Marco Roveri, and Paolo Busetta. Requirements-driven verification of web services. *Electr. Notes Theor. Comput. Sci.*, 105:95–108, 2004.

[PTBM05] Marco Pistore, Paolo Traverso, Piergiorgio Bertoli, and Annapaola Marconi. Automated synthesis of composite bpel4ws web services. In *IEEE International Conference on Web Services (ICWS)*, pages 293–301, 2005.

[PZQ+06] Geguang Pu, Huibiao Zhu, Zongyan Qiu, Shuling Wang, Xiangpeng Zhao, and Jifeng He. Theoretical foundations of scope-based compensable flow language for web service. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 4037 of *Lecture Notes in Computer Science*, pages 251–266, 2006.

[PZWQ06] Geguang Pu, Xiangpeng Zhao, Shuling Wang, and Zongyan Qiu. Towards the semantics and verification of BPEL4WS. *Electr. Notes Theor. Comput. Sci.*, 151(2):33–52, 2006.

[QWPZ05] Zongyan Qiu, Shuling Wang, Geguang Pu, and Xiangpeng Zhao. Semantics of bpel4ws-like fault and compensation handling. In *FM 2005: Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pages 350–365, 2005.

[SS04] K. Schmidt and C. Stahl. A petri net semantic for BPEL4WS validation and application. In *Proceedings of the 11th Workshop on Algorithms and Tools for Petri Nets (AWPN)*, pages 1–6, 2004.

[Tak08] Tsukasa Takemura. Formal semantics and verification of BPMN transaction and compensation. In *IEEE Asia-Pacific Services Computing Conference (APSCC)*, pages 284–290, 2008.

[Tha01] S. Thatte. XLANG — web services for business process design, 2001. Microsoft Corporation.

[vBK05] Franck van Breugel and Maria Koshika. Models and verification of BPEL, 2005. Available at http://www.cse.yorku.ca/ franck/research/-drafts/tutorial.pdf (Last accessed: 2010-02-11).

[vdA98] Wil M. P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.

[VFR09] Cátia Vaz, Carla Ferreira, and António Ravara. Dynamic recovering of long running transactions. *Trustworthy Global Computing: 4th International Symposium, TGC 2008, Barcelona, Spain, November 3-4, 2008, Revised Selected Papers*, pages 201–215, 2009.

[Vir04] Mirko Viroli. Towards a formal foundation to orchestration languages. *Electr. Notes Theor. Comput. Sci.*, 105:51–71, 2004.

[VvdA05] H. M. W. Verbeek and W.M.P. van der Aalst. Analyzing BPEL processes using petri nets. In *Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management*, pages 59–78, 2005.

[WDW07] Matthias Weidlich, Gero Decker, and Mathias Weske. Efficient analysis of BPEL 2.0 processes using p-calculus. In *Proceedings of The 2nd IEEE Asia-Pacific Services Computing Conference, APSCC 2007*, pages 266–274, 2007.

[WFN04] Andreas Wombacher, Peter Fankhauser, and Erich J. Neuhold. Transforming BPEL into annotated deterministic finite state automata for service discovery. In *Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, pages 316–323, 2004.

[WG08] Peter Y. H. Wong and Jeremy Gibbons. A process semantics for BPMN. In *Formal Methods and Software Engineering, 10th International Conference on Formal Engineering Methods, ICFEM 2008*, volume 5256 of *Lecture Notes in Computer Science*, pages 355–374, 2008.

[yLsL09] Hui yun Long and Jian shi Li. A process algebra approach of BPEL4WS. *Information and Computing Science*, 4(2):93–98, 2009.

[YTX+06] Yanping Yang, QingPing Tan, Yong Xiao, Feng Liu, and Jinshan Yu. Transform BPEL workflow into hierarchical CP-nets to make tool support for verification. In *Frontiers of WWW Research and Development (APWeb)*, volume 3841 of *Lecture Notes in Computer Science*, pages 275–284, 2006.