An Operational Semantics for StAC, a Language for Modelling Long-running Business Transactions

Michael Butler[1] and Carla Ferreira[2]

[1] School of Electronics and Computer Science
University of Southampton
Highfield, Southampton SO17 1BJ, United Kingdom
mjb@ecs.soton.ac.uk

[2] Department of Computer Science
Technical University of Lisbon
Av. Rovisco Pais, 1049-001 Lisbon, Portugal
carla.ferreira@dei.ist.utl.pt

Abstract. This paper presents the StAC language and its operational semantics. StAC (Structured Activity Compensation) is a business process modelling language and a distinctive feature of the language is its support for compensation. A compensation is an action taken to recover from error or cope with a change of plan, especially when rollback of a process is not possible. StAC is similar to a process algebraic language such as Hoare's CSP or Milner's CCS but has additional operators dealing with compensation and with exception handling. We have developed an operational semantics for the language which is presented in this paper.

1 Introduction

The StAC language (**St**ructured **A**ctivity **C**ompensation) was introduced in [3] as a business process modelling language and includes constructs for modelling compensation in business processes. In the context of business transactions, Gray [11] defines a compensation as the action taken to recover from error or cope with a change of plan. Compensation is a useful feature when modelling long running business transactions where rollback is not always possible because parts of a transaction will have been committed or because parts of a transaction (e.g., communications with external agents) are inherently impossible to undo. Consider the following example: a client buys some books in an on-line bookstore and the bookstore debits the client's account as the payment for the book order. The bookstore later realises that one of the books in the client's order is out of print. To compensate the client for this problem, the bookstore can credit the account with the amount wrongfully debited and send a letter apologising for their mistake. This example shows that compensation is more general than traditional rollback in database transactions. Compensation is important when a system cannot control everything, such as when interaction with humans is involved.

StAC gives a precise interpretation to the mechanics of compensation, including the combination of compensation with parallel execution, hierarchy and exceptions. StAC was inspired by the BPBeans framework [6] that allows an application to be built by a nested hierarchy of business processes. Like BPBeans, StAC provides ways to coordinate basic acivities by supporting sequential and concurrent processes, as well as compensation and exceptions. Similar coordination mechanisms are found in the BizTalk [15] and BPEL4WS [7] business coordination languages.

2 The StAC Language

StAC has taken inspiration from other process algebras, especially Milner's Calculus for Communicating Systems (CCS) [16] and Hoare's Communicating Sequential Processes (CSP) [12]. Both CCS and CSP model processes in terms of the atomic events in which they can engage. The languages provide operators for defining structured processes such as sequencing, choice, parallel composition, communication and hiding. StAC provides a similar process term language along with operators for compensation and exceptions. A StAC process has global data state associated with it and this state is changed by atomic events (or activities). Typically the data states of a StAC process are represented using state variables and the effect of atomic activities is represented using assignment to variables.

Formally a system is described by a set of equations of the form

$$N_i = P_i,$$

where each N_i is a process name and P_i is a StAC process expression. The syntax of StAC processes is presented in Table 1. Note that recursion is allowed since a process P_i may contain a call to a process named N_j.

The specification of a system is not complete with the StAC equations alone, as we might also want to specify the effect of the basic activities on the information structures. Instead of extending StAC to include variables and expressions we use existing state-based formal notations to define the state variables and activities. Possible notations are Z [19], VDM [13], B [1] and the guarded command language [8]. Any formal notation where it is possible to define a state, and where operations are partial relations on that state may be used to complement StAC. Examples of the use of the B notation [1] to specify the data states of a StAC process and the effect of activities on the variables may be found in [3, 9, 10].

Process ::=	A	(activity label)
	$\mid$ *skip*	(skip)
	$\mid b \;\&\; P$	(condition)
	$\mid call(N)$	(call named process)
	$\mid P \setminus S$	(hiding)
	$\mid P; Q$	(sequence)
	$\mid P \underset{X}{\parallel} Q$	(parallel)
	$\mid P \mathbin{[\!]} Q$	(external choice)
	$\mid P \sqcap Q$	(internal choice)
	$\mid P\{Q\}R$	(attempt block)
	$\mid \odot$	(early termination of attempt)
	$\mid P \div Q$	(compensation pair)
	$\mid \boxtimes$	(reverse)
	$\mid$ ☑	(accept)
	$\mid [P]$	(compensation scoping)

Table 1. StAC Syntax

2.1 StAC Operators

The StAC language allows sequential and parallel composition of processes, and the usual process combinators. Besides these, it has specific combinators to deal with compensation. An overview of the language is given in this section.

Each activity label A (in StAC) has an associated activity $\xrightarrow{A}$ representing an atomic change in the state: if Σ is the set of all possible states, then $\xrightarrow{A}$ is a relation on Σ.

The process *skip* does nothing and immediately terminates successfully. This process has a similar interpretation to the CSP *skip* that describes successful termination. The process call $call(N)$ calls the process named N returning if and when N terminates successfully.

Hiding is identical to the CSP hiding operator. With hiding one can make the execution of activities invisible to the environment.

Sequential and Parallel Operators. The sequential construct combines two processes, $P; Q$. In process $P; Q$, P is executed first, and only when P terminates successfully can Q be executed.

In parallel process $P \underset{X}{\parallel} Q$, the execution of the activities of P and Q is synchronised over the activities in X, while the execution of the remaining activities of P and Q is interleaved. Synchronisation can introduce deadlock, *e.g.*, process P may be waiting to synchronise with Q over an activity A and that activity will never occur in Q. If set X is empty (no synchronisation) we will represent the parallel process as $P \parallel Q$.

Condition. In the conditional operator, process P is guarded by a boolean function b. This boolean function can consult the state, *i.e.*, $b : \Sigma \rightarrow \mathbf{BOOL}$. Process $b \ \& \ P$ behaves as P if b evaluates to true in the current state. Conversely, if b is false, the conditional process will block.

Choice. The external choice $P \mathbin{[]} Q$ selects whichever of P or Q is enabled (i.e., not blocked). If both P and Q are enabled, the choice is made by the environment and it is resolved at the first activity. The environment could be a user selecting one of the options in a menu, for example. Notice that the $[]$ operator causes nondeterminism in some cases. Consider the following example:

$$(A; B) \mathbin{[]} (A; C).$$

When activity A occurs it is not possible to determine which one of the two behaviours $A; B$ or $A; C$ will be chosen. In this case, the choice is made internally by the system rather than by the environment. Internal nondeterminism may be specified directly using the internal choice operator ($\sqcap$).

The parallel and choice operators may be extended to generalised versions over (possibly infinite) sets of indexed processes. For example, a process that allows a user to choose a book could be described in StAC as:

$$[] \, b \in \mathit{BOOK} \bullet \mathit{ChooseBook}.b$$

Details of the generalised versions of the operators may be found in [9]

Attempt Block and Early Termination. An important feature in business processing is the possibility of terminating processes before they have concluded their main tasks. Early termination might arise if an exception occurs or a customer decides to abandon a transaction. It might also arise in the case of speculative parallelism, where several tasks, representing alternative ways of achieving a goal, are commenced in parallel and when one completes, the remaining tasks may be abandoned. We have included in StAC what we term an *attempt* block. An attempt block $P\{Q\}R$ first executes Q, and if Q terminates successfully it then continues with P. If an early termination operation ($\odot$) is executed within Q, the block continues with R. For example, the process

$$C\{A; \odot; B\}D$$

will first execute A, then the early termination will cause B to be skipped over and D to be executed. Any behaviour sequentially following the execution of early termination within an attempt block will be skipped. So an attempt block $P\{Q\}R$ can be viewed as an exception construct, with early termination representing the raising of an exception and R representing the exception handler.

The effect of the early termination is limited to the attempt block so in the following process, the early termination in the attempt block has no effect on the process S running in parallel with the block:

$$\{(P; \odot; Q)\} \parallel S$$

(We write $\{Q\}$ as short for $\mathit{skip}\{Q\}\mathit{skip}$.)

In the case of parallel processes within an attempt block, a termination instruction within one of the parallel process also affects the other processes. For example, in the process

$$\{\ (P; \odot; Q) \parallel R\ \}$$

the early termination after P allows R to terminate early. Our use of the term 'allows' is deliberate here. R is not required to terminate immediately. It may continue for several more steps before terminating early, it may continue to completion or it may execute forever if it is a non-terminating process. In any case, if and when the main body of an attempt block terminates and at least one of its constituent process has executed an early termination, then the whole of the main body is deemed to have terminated early.

Compensation Operators. The next few StAC operators are related to compensation. In the *compensation pair* $P \div Q$, P is the primary process and Q is the compensation process. When a compensation pair runs, it runs the primary task, and once the primary process has successfully completed, the compensation process is remembered (installed) for possible later invocation.

The *reverse* operation ($\boxtimes$) causes the currently installed compensation handlers to be invoked. For example

$$(A \div A'); \boxtimes$$

will execute A, install A' and then the reverse operation will cause A' to be executed. The overall behaviour is $A; A'$.

In the case of activities composed using sequential composition, the compensation process is constructed in the reverse order to the primary process execution. Consider the following process:

$$(A \div A'); (B \div B').$$

This process behaves as $A; B$ and has the compensation task $B'; A'$. A sequential compensation task can be viewed as a stack where compensation processes are pushed on to the top of the stack. The process

$$(A \div A'); (B \div B'); \boxtimes$$

behaves as $A; B$, and then the $\boxtimes$ operator causes the compensation task to be executed, so the overall behaviour is $(A; B); (B'; A')$ (which we write as $A; B; B'; A'$).

In the case of parallel processes, execution of compensations is also performed in parallel. The parallel process

$$(A \div A')\ \parallel\ (B \div B')$$

executes A and B concurrently and the resulting compensation process is $A' \parallel B'$.

The *accept* operation ($\boxdot$) indicates that currently installed compensations should be cleared, meaning that after an accept the compensation task is set to *skip*. The process

$$(A \div A'); (B \div B'); \boxdot; \boxtimes$$

executes A and B, when the $\boxtimes$ operation is called the compensation task $B'; A'$ has already been cleared by the $\boxdot$ operator so $B'; A'$ will not be executed.

Next we will consider the combination of compensation with choice. The process

$$(A \div A') \mathbin{[\,]} (B \div B')$$

behaves as either A or B, the choice between A and B is made by the environment. The compensation task in the case that A is chosen is A' and in the other case is B'.

If the primary process terminates early, the compensation process will not be installed. For example, in the process:

$$(A; \odot; B) \div C$$

compensation C would not be installed because of the early termination in the primary process.

In the case that a compensation pair is running in parallel with a process that executes an early termination, this early termination cannot affect the compensation pair while the compensation pair is executing. So the compensation pair will either not get executed at all or will be expected to execute to completion, including installation of the compensation handler. For example, the process

$$\{\ (A \div B) \parallel \odot\ \}$$

will either behave as *skip* or as $(A \div B)$.

The StAC language permits nested compensation pairs, meaning that compensation can itself be compensated. The following process has two levels of compensation:

$$A \div (B \div C).$$

Initially the above process behaves as A and the compensation task $B \div C$ is remembered as the compensation for A. When the reverse operator is appended to the previous process

$$(A \div (B \div C)); \boxtimes$$

after the execution of activity A the reversal will cause compensation pair $B \div C$ to be executed, by executing B and adding C to the compensation. Activity C can be invoked later by a reversal to compensate for activity B. The nested compensation pair states that A is compensated by B, and B is compensated by C.

Scoping of Compensation. The compensation scoping brackets $[\cdots]$ provide nested compensation scoping are used to delimit the scope of the acceptance and reversal operators. All StAC processes have an implicit outer compensation scope. The start of scope creates a new compensation task, and invoking a reversal instruction within that scope will only execute those compensation activities that have been remembered since the start of the scope. In the process

$$(A \div A'); [\,(B \div B'); \boxtimes\,],$$

the overall process would behave as $A; B; B'$. Compensation A' is not invoked because its outside the scope of the reversal instruction. An acceptance instruction, within a scope, will only clear the compensation activities that have been recorded since the start of the scope. For example, the process:

$$(A \div A'); [\,(B \div B'); \boxdot\,]; (C \div C')$$

after A, B and C have been executed, has $C'; A'$ as compensation. Since the acceptance instruction is within the compensation scope, it just clears the compensation process B' that is within the brackets. Another feature of compensation scoping is that compensation is remembered beyond a scope if a reversal instruction is not performed, as in the example:

$$(A \div A'); [\,(B \div B')\,]; (C \div C').$$

Here, after executing $A; B; C$ the compensation process is $C'; B'; A'$, which includes the compensation process B' of the inner scope. B' is retained because there is no acceptance instruction within the brackets.

2.2 Example: Order Fulfillment

To illustrate the use of StAC we present the order fulfillment example described in [4] and [5]. ACME Ltd distributes goods which have a relatively high value to its customers. When the company receives an order from a customer, the first step is to verify whether the stock is available. If not available the customer is informed that his/her order can not be accepted. Otherwise, the warehouse starts preparing the order for shipment, and a courier is booked to deliver the goods to the customer. Simultaneously with the warehouse preparing the order, the company does a credit check on the customer to verify that the customer can pay for the order. The credit check is performed in parallel because it normally succeeds, and in this normal case the company does not wish to delay the order unnecessarily. If the credit check fails the preparation of the order is stopped. Here we present a very simple representation of the order acceptance and focus on the order fulfillment part in more detail.

Before presenting the ACME process, we introduce the following syntactic sugar:

$$\begin{aligned} \mathit{TRY}\ P\ \mathit{THEN}\ Q\ \mathit{ELSE}\ R &= Q\{P\}R \\ \mathit{IF}\ G\ \mathit{THEN}\ P\ \mathit{ELSE}\ Q &= G\&P \ [] \ \neg G\&Q \end{aligned}$$

At the top level the application is defined as a sequence as follows:

$$\begin{aligned} \mathit{ACME} = {} & \mathbf{AcceptOrder} \div \mathbf{RestockOrder}; \\ & \mathit{TRY}\ \mathit{FulfillOrder}\ \mathit{THEN}\ \boxdot\ \mathit{ELSE}\ \boxtimes \end{aligned}$$

The first step in the *ACME* process is a compensation pair. The primary action of this pair is to accept the order and deduct the order quantity from the inventory database. The compensation action is simply to add the order quantity back to the total in the inventory database. Following the compensation pair, the *FulfillOrder* process is invoked. If the order has been fulfilled correctly (*FulfillOrder* terminates sucessfully), the order is accepted, otherwise (*FulfillOrder* terminates early) the order is reversed.

Notice that some processes are written with a bold font, e.g., **AcceptOrder**, this means that those processes are activity labels, so they are not further decomposed.

The order is fulfilled by packaging the order at the warehouse while concurrently doing a credit check on the customer. If the credit check fails, the *FulfillOrder* process is terminated early:

$$\begin{aligned} \mathit{FulfillOrder} = {} & \mathit{WarehousePackaging} \\ & \| \ (\mathbf{CreditCheck}; \mathit{IF}\ \neg\mathbf{okCreditCheck}\ \mathit{THEN}\ \odot\ \mathit{ELSE}\ \mathit{skip}\ ) \end{aligned}$$

(IF B THEN P ELSE Q is short for $(B \ \& \ P) \ [] \ (\neg B \ \& \ Q)$.) Because *WarehousePackaging* is within the scope of the early termination, a failed credit check allows *WarehousePackaging* to terminate early, possible before all the items in the order have been packed.

The *WarehousePackaging* process consists of a compensation pair in parallel with the *PackOrder* process:

$$\mathit{WarehousePackaging} = (\mathbf{BookCourier} \div \mathbf{CancelCourier}) \parallel \mathit{PackOrder}$$

The compensation pair books the courier, with the compensation action being to cancel the courier booking. **CancelCourier** might result in a second message being sent to the courier rather than reversing the send of the message which booked the courier. The *PackOrder* process packs each of the items in the order in parallel. Each **PackItem** activity is reversed by a corresponding **UnpackItem**:

$$\mathit{PackOrder} \ = \| i \in \mathbf{order} \ \bullet \ (\mathbf{PackItem}(i) \div \mathbf{UnpackItem}(i))$$

In the case that a credit check fails, the *FulfillOrder* process terminates early with the courier possibly having been booked and possibly some of the items having being packed. The reversal instruction will then be invoked and will result in the appropriate compensation activity being invoked for those activities that did take place.

3 Semantics

In this section we will present the operational semantics for StAC. To do this we introduce a variant of the language called StAC$_i$ to which we give an operational semantics. We refer to StAC$_i$ as the semantic language. In StAC$_i$, a process can have several simultaneous compensation tasks associated with it. A process decides which task to attach the compensation activities to, and each individual compensation task can be reversed or accepted. This contrasts with the language presented in Section 2, where scoping of compensation is hierarchical and each scope has a single implicit compensation task. To distinguish different compensation tasks, the operators that deal with compensation, *i.e.*, compensation pair, acceptance and reversal, are indexed by the compensation task index to which they apply. The syntax of StAC$_i$ is presented in Table 2. Later we define a translation from StAC to StAC$_i$.

Process ::=	A	(activity label)
	$\mid$ *skip*	(skip)
	$\mid b \mathbin{\&} P$	(conditional)
	$\mid call(N)$	(named process call)
	$\mid P \setminus S$	(hiding)
	$\mid P; Q$	(sequence)
	$\mid P \underset{X}{\parallel} Q$	(parallel)
	$\mid P \sqcap Q$	(internal choice)
	$\mid P \mathbin{[\,]} Q$	(external choice)
	$\mid \odot$	(early termination of attempt)
	$\mid P\{Q\}_v R$	**(attempt block)**
	$\mid$ *early*	**(prematurely terminated process)**
	$\mid \lvert P \rvert_v$	**(protection block)**
	$\mid new(i).P_i$	**(create new compensation task)**
	$\mid \boxtimes_i$	**(indexed reverse)**
	$\mid$ ☑$_i$	**(indexed accept)**
	$\mid \uparrow_i P$	**(push)**
	$\mid J \rhd i$	**(merge)**

Table 2. StAC$_i$ Syntax ($[P]$ and $P \div Q$ are derived operators)

3.1 Semantic Language

Several of the StAC$_i$ operators are retained from StAC without any alterations. The changes concern operators that deal with compensation and early termination (indicated with bold font in Table 2).

After an early termination, the process within an attempt block may continue to execute for several steps before terminating. To deal with this we have added a boolean flag v to the attempt block representing the following two possibilities:

$P\{Q\}_{false}\, R$ – Process Q can continue its execution since no early termination instruction has been invoked within Q.

$P\{Q\}_{true}\, R$ – An early termination instruction has previously been invoked within the attempt block and process Q may terminate prematurely.

The term *early* represents a process that has terminated early. It is used to distinguish a process that has terminated early from a process that has terminated successfully (*skip*).

In some case we require that a process that has already commenced execution, e.g., a compensation pair, be protected from early termination caused by another process within an attempt block. This is achieved using the protection block $|P|_v$. The boolean flag v will initially be *false*, and once the protected process executes its first visible activity the flag will be changed to *true* and P will then be protected from an early termination originating from outside the protection block. A protection block may still be terminated by an early termination invocation within the protection block.

In our semantics, the compensation information of a process is maintained by a compensation function that maps each compensation task index to a compensation process. The accept and reverse operators are subscripted with the index of the compensation process to which they should be applied. In the StAC$_i$ language we introduce the *push* operator that stores a compensation process in a compensation task, i.e., $\uparrow_i Q$ will store Q on top of compensation task i, where i is an index. A compensation pair $P \div_i Q$ can be defined in terms of the push operator as follows:

$$|\ P\,;\ \uparrow_i Q\ |_{false}.$$

Here, P will be executed first and after it has concluded its execution the process *push* will store Q in task i. The protection block ensures that, once the process $P\,;\uparrow_i Q$ has commenced, it is not affected by early termination emanating from outside the protection block.

An important operator in StAC$_i$ is the merge operator. The expression $J \rhd i$, where J is a set of indices, merges all compensation tasks belonging to J into the compensation task i. All compensation tasks in J are put in parallel, the result is added to the compensation task i and all compensation tasks J are cleared. For example, in the process

$$(A \div_i A'); (B \div_j B'); \{i, j\} \rhd k$$

the merge operator will compose in parallel the compensation task i (A') with compensation task j (B'), and add parallel process $A' \parallel B'$ to compensation task k. Compensation tasks i and j will be removed.

Consider the following process that uses three individual compensation tasks:

$$(A \div_i A'); (B \div_j B'); (C \div_k C'); \{i, j\} \rhd k.$$

Initially it executes A, B and C and then merges compensation tasks i and j into compensation task k. Joining compensation tasks i and j results in the parallel process $A' \parallel B'$, that will be put in front of the compensation task k, giving $(A' \parallel B'); C'$ as the resulting compensation for task k.

The $new(i).P_i$ construct in StAC$_i$ creates a new compensation task identified by bound variable i. This can be used to model the compensation scope of StAC. For example, the StAC process

$$(A \div A'); [(B \div B'); \boxtimes; (C \div C')]$$

can be represented in StAC$_i$ as

$$(A \div_i A'); new(j).((B \div_j B'); \boxtimes_j; (C \div_j C'); \{j\} \rhd i)$$

When the reversal instruction is invoked on compensation task j it will only execute B'. Compensation process A' that in StAC was outside the scope, in StAC$_i$ is in a different compensation task, and does not get invoked. The merge is used to preserve any compensations not reversed within the scoping brackets.

3.2 Operational Semantics for Compensation

This section presents the operational semantics for the StAC$_i$ operators excluding termination operators. The semantics of termination is described in Section 3.3. Plotkin [18] describes how to use transition systems to define an operational semantics; here a system is defined in terms of transition rules between configurations. For the operational semantics of StAC$_i$, a configuration is a tuple:

$$(P, C, \sigma)\ \in\ Process \times (I \to Process) \times \Sigma$$

In the above tuple, C is a function that returns the compensation process $C(i)$, for each compensation index i. If $C(i) = \perp$, then the compensation is not in use. Σ represents the data state and Σ is included in our model of StAC processes since we want to model the ability of a basic activity to change the data state. The labelled transition

$$(P,\ C,\ \sigma) \xrightarrow{A} (P',\ C,\ \sigma') \qquad (1)$$

denotes that the execution of a basic activity A may cause a configuration transition from (P, C, σ) to (P', C, σ'). Notice that the execution of a basic activity does not alter the compensation function. Instead, only the compensation operators may alter the compensation function.

In the configuration transition (1) we used an activity as the transition label, but two other labels may be used, they are τ and $\odot$. The set $\mathcal{B}$ of all possible transition labels is defined as:

$$\mathcal{B} = \mathcal{A} \cup \{\tau, \odot\}$$

where $\mathcal{A}$ represents the set of all activity labels. The label τ is a special label that represents an operation not visible to the external environment. In the transition rules we consider that label $B \in \mathcal{B}$, while $A \in \mathcal{A}$.

Activity. We assume that an activity is a relation from states to states, and write $\sigma \xrightarrow{A} \sigma'$ when σ is related to σ' by $\xrightarrow{A}$. The execution of an activity imposes a change in the state, leaving the compensation function unchanged:

$$\frac{\sigma \xrightarrow{A} \sigma'}{(A,\ C,\ \sigma) \xrightarrow{A} (skip,\ C,\ \sigma')}$$

Conditional. In the conditional process $b \,\&\, P$ the execution of P is guarded by a boolean function b. If the boolean function b is true in the current state σ, then P may be executed:

$$\frac{(P,\ C,\ \sigma) \xrightarrow{B} (P',\ C',\ \sigma') \ \wedge\ b(\sigma) = true}{(b \,\&\, P,\ C,\ \sigma) \xrightarrow{B} (P',\ C',\ \sigma')}$$

Notice that when B is an activity, the guard is evaluated in the state to which the activity B is applied, ensuring that b holds when the activity is executed. Since there is no transition rule dealing with a false guard, a false guard causes the process to block.

Name Process Call. The call of a process N (where $N = P$ is an equation) will substitute $call(N)$ by the process on the left-side of the equation:

$$\frac{N = P}{(call(N),\ C,\ \sigma) \xrightarrow{\tau} (P,\ C,\ \sigma)}$$

Hiding. The rule on the left below says that the hiding operator makes the occurrence of an activity labelled from set S invisible to the environment. The rule on the right states that the occurrence of an activity not labelled from S is visible:

$$\frac{(P,\ C,\ \sigma) \xrightarrow{A} (P',\ C',\ \sigma') \ \wedge\ A \in S}{(P \setminus S,\ C,\ \sigma) \xrightarrow{\tau} (P' \setminus S,\ C',\ \sigma')} \qquad \frac{(P,\ C,\ \sigma) \xrightarrow{B} (P',\ C',\ \sigma') \ \wedge\ B \notin S}{(P \setminus S,\ C,\ \sigma) \xrightarrow{B} (P' \setminus S,\ C',\ \sigma')}$$

Hiding events in *skip* is the same as *skip*:

$$\frac{}{(skip \setminus S,\ C,\ \sigma) \xrightarrow{\tau} (skip,\ C,\ \sigma)}$$

Sequence. The next rule shows the execution of activities within the first process of a sequential composition:

$$\frac{(P,\ C,\ \sigma) \xrightarrow{B} (P',\ C',\ \sigma')}{(P;\ Q,\ C,\ \sigma) \xrightarrow{B} (P';\ Q,\ C',\ \sigma')}$$

If the first process in the sequence has terminated successfully, then the second process can be executed immediately:

$$\frac{}{(skip;\ Q,\ C,\ \sigma) \xrightarrow{\tau} (Q,\ C,\ \sigma)}$$

Parallel. The rule below shows that the occurrence of activity $A \in X$ requires processes P and Q to synchronise in $P \underset{X}{\parallel} Q$:

$$\frac{A \in X \;\;\wedge\;\; (P,\, C,\, \sigma) \xrightarrow{A} (P',\, C,\, \sigma') \;\;\wedge\;\; (Q,\, C,\, \sigma) \xrightarrow{A} (Q',\, C,\, \sigma')}{(P \underset{X}{\parallel} Q,\, C,\, \sigma) \xrightarrow{A} (P' \underset{X}{\parallel} Q',\, C,\, \sigma')}$$

Notice that both processes refer to the same basic activity which updates the state from σ to σ'. This is because the state is intended to be global and the parallel processes do not have their own local state.

The following two rules specify that parallel processes can evolve independently for $B \notin X$:

$$\frac{B \notin X \;\;\wedge\;\; (P,\, C,\, \sigma) \xrightarrow{B} (P',\, C',\, \sigma')}{(P \underset{X}{\parallel} Q,\, C,\, \sigma) \xrightarrow{B} (P' \underset{X}{\parallel} Q,\, C',\, \sigma')} \qquad \frac{B \notin X \;\;\wedge\;\; (P,\, C,\, \sigma) \xrightarrow{B} (P',\, C',\, \sigma')}{(Q \underset{X}{\parallel} P,\, C,\, \sigma) \xrightarrow{B} (Q \underset{X}{\parallel} P',\, C',\, \sigma')}$$

The rule below states that the parallel process $P \underset{X}{\parallel} Q$ terminates (i.e., reduces to *skip*) when both P and Q terminate:

$$\frac{}{(\mathit{skip} \underset{X}{\parallel} \mathit{skip},\, C,\, \sigma) \xrightarrow{\tau} (\mathit{skip},\, C,\, \sigma)}$$

Internal Choice. Internal choice decides nondeterministically which process P or Q will occur:

$$\frac{}{(P \sqcap Q,\, C,\, \sigma) \xrightarrow{\tau} (P,\, C,\, \sigma)} \qquad \frac{}{(P \sqcap Q,\, C,\, \sigma) \xrightarrow{\tau} (Q,\, C,\, \sigma)}$$

External Choice. The next two rules state that in $P \,\square\, Q$ only one of the processes P or Q is executed:

$$\frac{(P,\, C,\, \sigma) \xrightarrow{B} (P',\, C',\, \sigma') \;\;\wedge\;\; B \neq \tau}{(P \,\square\, Q,\, C,\, \sigma) \xrightarrow{B} (P',\, C',\, \sigma')} \qquad \frac{(P,\, C,\, \sigma) \xrightarrow{B} (P',\, C',\, \sigma') \;\;\wedge\;\; B \neq \tau}{(Q \,\square\, P,\, C,\, \sigma) \xrightarrow{B} (P',\, C',\, \sigma')}$$

The following rules state that the occurrence of internal actions will not resolve the choice between P and Q:

$$\frac{(P,\, C,\, \sigma) \xrightarrow{\tau} (P',\, C',\, \sigma')}{(P \,\square\, Q,\, C,\, \sigma) \xrightarrow{\tau} (P' \,\square\, Q,\, C',\, \sigma')} \qquad \frac{(Q,\, C,\, \sigma) \xrightarrow{\tau} (Q',\, C',\, \sigma')}{(P \,\square\, Q,\, C,\, \sigma) \xrightarrow{\tau} (P \,\square\, Q',\, C',\, \sigma')}$$

Create Compensation Task. The rule for the *new* construct selects an index for a compensation task that is not in use and sets that task to *skip*:

$$\frac{C(k) = \perp}{(\mathit{new}(i).P_i,\, C,\, \sigma) \xrightarrow{\tau} (P_k,\, C[k := \mathit{skip}],\, \sigma)}$$

$C[k := \mathit{skip}]$ denotes that compensation task k is set to *skip*

Push. The rule for the *push* operator adds the compensation process Q to the compensation function C:

$$\frac{}{(\uparrow_i Q,\, C,\, \sigma) \xrightarrow{\tau} (\mathit{skip},\, C[i := (Q; C(i))],\, \sigma)}$$

$C[i := Q; C(i)]$ denotes that compensation task i is set to Q in sequence with the previous compensation for task i. In this manner, the compensation process is built in the reverse order of the execution of the primary processes.

Reverse. In the next rule, the operator $\boxtimes_i$ causes the compensation task i to be executed, and also resets that compensation task to *skip*:

$$\frac{}{(\boxtimes_i,\ C,\ \sigma)\ \xrightarrow{\tau}\ (C(i),\ C[i := \mathit{skip}],\ \sigma)}$$

Note that compensation tasks do not store any state with them: if the state changes between the compensation being stored and executed, the current state is used.

Accept. The operator $\boxdot_i$ clears the compensation task i to *skip* without executing it:

$$\frac{}{(\boxdot_i,\ C,\ \sigma)\ \xrightarrow{\tau}\ (\mathit{skip},\ C[i := \mathit{skip}],\ \sigma)}$$

Merge. The operator $J \rhd i$ merges all compensation tasks of set J in parallel on to the front of compensation task i:

$$\frac{}{(J \rhd i,\ C,\ \sigma)\ \xrightarrow{\tau}\ (\mathit{skip},\ C[\ \ i := (||_{j \in J} . C(j)); C(i),\quad J :=\perp\ \ ],\ \sigma)}$$

In the above rule the expression $J\ :=\perp$ denotes attributing to all tasks of set J the value $\perp$ meaning these tasks are no longer in use. Set J must be disjoint from i.

3.3 Operational Semantics for Termination

This section concludes the presentation of StAC$_i$ semantics by defining the operational rules related to early termination.

Protected Block. This rule states that the occurrence of a basic activity within a protected process P will place the label *true* on the protection block. It is not necessary to distinguish whether the value v is initially *true* or *false*, in both cases the final label will be *true*:

$$\frac{(P,\ C,\ \sigma)\ \xrightarrow{B}\ (P',\ C',\ \sigma')\quad \wedge\quad B \neq \tau\quad \wedge\quad v \in \mathit{BOOL}}{(\ |P|_v\ ,\ C,\ \sigma\ )\ \xrightarrow{B}\ (\ |P'|_{\mathit{true}}\ ,\ C',\ \sigma'\ )}$$

Note that occurrence of a τ is not regarded as commencing a protection block so the above rule does not apply to τ. The following rule deals with τ:

$$\frac{(P,\ C,\ \sigma)\ \xrightarrow{\tau}\ (P',\ C',\ \sigma')\quad \wedge\quad v \in \mathit{BOOL}}{(\ |P|_v\ ,\ C,\ \sigma\ )\ \xrightarrow{\tau}\ (\ |P'|_v\ ,\ C',\ \sigma'\ )}$$

A terminated protection block becomes *skip*:

$$\frac{}{(\ |\mathit{skip}|_v\ ,\ C,\ \sigma\ )\ \xrightarrow{\tau}\ (\ \mathit{skip}\ ,\ C,\ \sigma\ )}$$

Early Termination. Invocation of the early termination operation causes a process to execute a visible $\odot$-event and then become the *early* process:

$$\frac{}{(\odot,\ C,\ \sigma)\ \xrightarrow{\odot}\ (\mathit{early},\ C,\ \sigma)}$$

Later, when the rules for the attempt block are presented, it will be seen that this early termination event will cause the enclosing attempt block to commence early termination and that the event will be contained within the enclosing attempt block and will not be visible outside it.

In a sequential process if the first process has terminated early, the overall sequential process is interrupted and terminates early:

$$\frac{}{(\mathit{early};\ P,\ C,\ \sigma)\ \xrightarrow{\tau}\ (\mathit{terminate}(P); \mathit{early},\ C,\ \sigma)}$$

The *terminate* function terminates all constituents of P except for compensation merge operations and protected blocks that have commenced execution. The *early* process is added to ensure that the indication of early termination is maintained. Allowing the merge to be executed is important because merging of compensations will typically be required before exiting an attempt block. The *terminate* function is defined later in this section.

A consequence of the previous rule is that if in a compensation pair the primary process terminates early, the overall process terminates early and the compensation process Q will not be installed.

Hiding or protection of an early terminated process gives an early terminated process:

$$\frac{}{(\ early \setminus S\ ,\ C,\ \sigma\ )\ \xrightarrow{\tau}\ (\ early\ ,\ C,\ \sigma\ )} \qquad \frac{}{(\ |early|_v\ ,\ C,\ \sigma\ )\ \xrightarrow{\tau}\ (\ early\ ,\ C,\ \sigma\ )}$$

In a parallel process if both have terminated early, or one has terminated early and the other has terminated early, the overall process will terminate early:

$$\frac{}{(early \underset{X}{\parallel} early,\ C,\ \sigma)\ \xrightarrow{\tau}\ (early,\ C,\ \sigma)}$$

$$\frac{}{(early \underset{X}{\parallel} skip,\ C,\ \sigma)\ \xrightarrow{\tau}\ (early,\ C,\ \sigma)} \qquad \frac{}{(skip \underset{X}{\parallel} early,\ C,\ \sigma)\ \xrightarrow{\tau}\ (early,\ C,\ \sigma)}$$

Attempt Block. If the main body of an attempt block can engage in an early termination event, then its termination flag is set to *true* and the early termination event is made invisible:

$$\frac{(Q,\ C,\ \sigma)\ \xrightarrow{\odot}\ (Q',\ C,\ \sigma)}{(P\{\ Q\ \}_v\, R,\ C,\ \sigma)\ \xrightarrow{\tau}\ (P\{\ Q'\ \}_{true}\, R,\ C,\ \sigma)}$$

When the main body of an attempt block terminates successfully, the left hand continuation process will be executed:

$$\frac{}{(P\{skip\}_v\, R,\ C,\ \sigma)\ \xrightarrow{\tau}\ (P,\ C,\ \sigma)}$$

(The flag v will always be $false$ whenever an attempt block body evolves to *skip*.)

When the main body of an attempt block terminates prematurely, the right hand continuation process will be executed:

$$\frac{}{(P\{early\}_v\, R,\ C,\ \sigma)\ \xrightarrow{\tau}\ (R,\ C,\ \sigma)}$$

An attempt block may also evolve by executing events of the main body other than $\odot$, regardless of the value of the flag v:

$$\frac{(Q,\ C,\ \sigma)\ \xrightarrow{B}\ (Q',\ C',\ \sigma')\ \ \wedge\ \ B \neq \odot}{(P\{Q\}_v\, R,\ C,\ \sigma)\ \xrightarrow{B}\ (P\{Q'\}_v\, R,\ C',\ \sigma')}$$

When the termination flag has been set to *true*, the main body of an attempt block may terminate early. However, all protected blocks that have started their execution are made to complete their execution. This is provided for in as follows:

$$\frac{}{(P\{Q\}_{true}\, R,\ C,\ \sigma)\ \xrightarrow{\tau}\ (P\{terminate(Q)\}_{false}\, R,\ C,\ \sigma)}$$

Note that setting the termination flag back to $false$ in this rule prevents *terminate* being applied infinitely which would otherwise cause an infinite cycle of τ events.

Function terminate. The *terminate* function clears all processes that no longer should continue and keeps the protected blocks that have already started. The first three definitions below show processes that may continue running, as they may contain protected blocks. The fourth rule says that a merge of compensation tasks is allowed to execute. The fifth rule shows that early termination is propagated by *terminate*:

$$\begin{array}{lcl} terminate(P;\,Q) & = & terminate(P); terminate(Q) \\ terminate(P \parallel_X Q) & = & terminate(P) \parallel_X terminate(Q) \\ terminate(P\{Q\}_v R) & = & \{terminate(Q)\}_v \\ terminate(J \rhd i) & = & J \rhd i \\ terminate(early) & = & early \end{array}$$

The next rule shows that a protected block that has not started its execution (its flag is *false*) is terminated immediately. The rule following that states that a protected block that has started its execution (its flag is *true*) can continue until it has finished:

$$\begin{array}{l} terminate(|P|_{false}) = skip \\ terminate(|P|_{true}) = |P|_{true} \end{array}$$

The final rule applies to any process not matching the previous rules for *terminate*. Such processes are made to finish immediately:

$$terminate(P) = skip$$

It is easy to show that *terminate* is idempotent.

3.4 Translation from StAC to StAC$_i$

We define a semantics for the StAC language by defining a translation of StAC processes into StAC$_i$ processes. This way the interpretation of a StAC process is given in terms of a StAC$_i$ process.

For each process definition of the form $N = P$, we construct an indexed process definition:

$$N_i \;=\; \mathbb{T}(P, i)$$

where $\mathbb{T}$ translates a process written in the syntax of StAC to a process in the syntax of StAC$_i$ in the context of compensation index i. The translation function $\mathbb{T}$ is shown in Table 3. For the translation to work correctly, the root StAC process definition should be of the form $N = [P]$, ensuring that the outermost compensation task is properly created.

The first three rules of the translation function $\mathbb{T}$ show that basic activities, *skip* and $\odot$ have the same representation in StAC and StAC$_i$. The next rule shows that a call to process named N in the context of index i becomes a call to process N_i.

The StAC$_i$ representation for the compensation operators accept, reverse, and compensation pair, is obtained by adding a compensation task index to each of them.

The remaining rules (except the last two), show how to translate composite constructs. The translation rules are defined on the constituents of the constructor.

Because the order of execution of $P \underset{X}{\parallel} Q$ is not known, their compensation should not have a predefined order of execution. So each parallel process will have a new compensation task. This way their compensation processes will also be a parallel process: the parallel composition of the new compensation tasks. In the translation rule for $P \parallel Q$, two new compensations tasks j and k are created and processes P and Q are translated using j and k. The resulting processes will be composed in parallel. Lastly, the new compensation tasks j and k are merged into the initial task i, which means that the compensations of the parallel processes are retained (unless they have been explicitly committed). Notice that compensation tasks are merged in parallel, so the outcome of the merge is process $C(j) \parallel C(k)$, that will be pushed on top of $C(i)$.

The compensation scoping brackets $[P]$ are translated to a new compensation task j and process P is translated using index j. Then the compensation task j is merged into the initial index i, so all the compensation information that was not reversed or accepted can be preserved by adding it to compensation task i.

$$
\begin{aligned}
&\mathbb{T}(A, i) &&= A \\
&\mathbb{T}(skip, i) &&= skip \\
&\mathbb{T}(\odot, i) &&= \odot \\
&\mathbb{T}(call(N), i) &&= call(N_i) \\
&\mathbb{T}(\boxtimes, i) &&= \boxtimes_i \\
&\mathbb{T}(☑, i) &&= ☑_i \\
&\mathbb{T}(P \div Q, i) &&= |\mathbb{T}(P, i) \ \div_i \ \mathbb{T}(Q, i)|_{false} \\
&\mathbb{T}(b \,\&\, P, i) &&= b \,\&\, \mathbb{T}(P, i) \\
&\mathbb{T}(P \setminus S, i) &&= \mathbb{T}(P, i) \setminus S \\
&\mathbb{T}(P; Q, i) &&= \mathbb{T}(P, i)\,;\,\mathbb{T}(Q, i) \\
&\mathbb{T}(P \,[]\, Q, i) &&= \mathbb{T}(P, i) \,[]\, \mathbb{T}(Q, i) \\
&\mathbb{T}(P \sqcap Q, i) &&= \mathbb{T}(P, i) \sqcap \mathbb{T}(Q, i) \\
&\mathbb{T}(P\{Q\}R, i) &&= \mathbb{T}(P, i)\{\mathbb{T}(Q, i)\}\mathbb{T}(R, i) \\
&\mathbb{T}([P], i) &&= new(j).(\ \mathbb{T}(P, j); \{j\} \rhd i\ ) \\
&\mathbb{T}(P \underset{X}{\parallel} Q, i) &&= new(j, k).(\ (\mathbb{T}(P, j) \underset{X}{\parallel} \mathbb{T}(Q, k)); \{j, k\} \rhd i\ )
\end{aligned}
$$

Table 3. Translation Rules

4 Conclusions

The semantic definition of StAC is somewhat complicated, in particular the use of indexed compensation tasks. An alternative approach would be to embed the installed compensations within the process terms, for example, by representing a scope in the form $[P, Q]$ where P is the remaining process to be executed within the scope and Q represents the compensation installed so far for this scope. However the interaction between early termination and compensation means that installed compensations must be preserved whenever a scope terminates early. Because scopes can be nested, this requires that the installed compensations for the nested scopes need to be accumulated before a process terminates. We found it easier to describe this by separating the installed compensations from the process terms which in turn required the use of indexed compensation.

In [5] we have used indexed compensation to explore generalisations of the modelling language in which processes may have multiple compensation 'threads'. It is not clear how useful this is in its full generality, but two idioms do appear to be useful: selective compensation, where some activities are compensated and others are not (yet) compensated, and alternative compensation, where activities can have several alternative compensations and the compensation to be selected may depend on the nature of the exception.

The use of indexed compensation should also make it possible to model the style of compensation used in BPEL4WS [7]. BPEL4WS supports similar operators to StAC, such as compensation, concurrency, and sequencing. In BPEL4WS, reversal is invoked through exception handlers, while acceptance is implicit in scoping. Reversal (called *compensate* in BPEL4WS) can identify particular sub-processes which should be compensated and this can be modelled using indexed compensation. BPEL4WS is layered on top of XML (its processes and data are specified in the BPEL dialect of XML), and at the moment BPEL4WS does not have a formal semantics. We plan to investigate further the use of StAC to give a semantics to BPEL4WS.

In [14], a compensation is formalised in terms of the properties it has to guarantee. However, [14] does not provide a modelling language as StAC does, rather it provides a characterisation of properties of compensation. Bocchi et al [2] define a language πt-calculus for modelling long-running transactions based on Milner's π-calculus [17]. The πt-calculus includes a transaction construct that contains a compensation handler and a fault manager. ConTracts [20] attempt to provide a structured approach to compensation. In ConTracts the invocation of a particular compensation has to be made explicitly within a conditional instruction (if the outcome of a step is false, then a specific task is executed to compensate for this). ConTracts do not have the notion of installing a compensation handler nor acceptance nor reversal found in StAC.

Acknowledgments

We would like to thank Peter Henderson, Mandy Chessell, David Vines and Catherine Griffin for helping us to understand compensation. Thanks to Tony Hoare and Viorel Preoteasa for useful comments on the language and its semantics. Thanks to the anonymous referees of COORD04 for many useful suggestions.

References

1. J.R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. L. Bocchi, C. Laneve, and G. Zavattaro. A calulus for long-running transactions. In *FMOODS'03*, volume LNCS, to appear. Springer-Verlag, 2003.
3. M. Butler and C. Ferreira. A process compensation language. In *Integrated Formal Methods(IFM'2000)*, volume 1945 of *LNCS*, pages 61 – 76. Springer-Verlag, 2000.
4. M. Chessell, D. Vines, and C. Griffin. An introduction to compensation with business process beans. Technical report, Transaction Processing Design and New Technology Development Group, IBM UK Laboratories, August 2001.
5. M. Chessell, D. Vines, C. Griffin, M. Butler, C. Ferreira, and P. Henderson. Extending the concept of transaction compensation. *IBM Systems Journal*, 41(4):743–758, 2002.
6. M. Chessell, D. Vines, C. Griffin, V. Green, and K. Warr. Business process beans: System design and architecture document. Technical report, Transaction Processing Design and New Technology Development Group, IBM UK Laboratories, January 2001.
7. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business process execution language for web services, version 1.1. `http://www-106.ibm.com/developerworks/library/ws-bpel/`, 2003.
8. E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
9. C. Ferreira. *Precise Modelling of Business Processes with Compensation*. PhD thesis, University of Southampton, 2002.
10. C. Ferreira and M. Butler. Using B Refinement to Analyse Compensating Business Processes. In *Third International ZB Conference (ZB'2003)*, volume 2651 of *LNCS*. Springer-Verlag, 2003.
11. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.
12. C.A.R Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
13. C. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 1986.
14. H. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *16th VLDB Conference*, Brisbane, Australia, 1990.
15. B. Metha, M. Levy, G. Meredith, T. Andrews, B. Beckman, J. Klein, and A. Mital. BizTalk Server 2000 Business Process Orchestration. *IEEE Data Engineering Bulletin*, 24(1):35–39, 2001.
16. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
17. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Inform. and Comput.*, 100(1):1–40,41–77, 1992.
18. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, September 1981.
19. J. Spivey. *The Z Notation*. Prentice Hall, New York, 1989.
20. H. Wachter and A. Reuter. The ConTract model. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, 1992.