Journal of Universal Computer Science, vol. 11, no. 5 (2005), 712-743
submitted: 30/11/04, accepted: 31/1/05, appeared: 28/5/05 © J.UCS

Precise Modelling of Compensating Business Transactions and its Application to BPEL

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

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

Muan Yong Ng
School of Electronics and Computer Science, University of Southampton
Highfield, Southampton SO17 1BJ, United Kingdom
myn@ecs.soton.ac.uk

Abstract: We describe the StAC language which can be used to specify the orchestration of activities in long running business transactions. Long running business transactions use compensation to cope with exceptions. StAC supports sequential and parallel behaviour as well as exception and compensation handling. We also show how the B notation may be combined with StAC to specify the data aspects of transactions. The combination of StAC and B provides a rich formal notation which allows for succinct and precise specification of business transactions. BPEL is an industry standard language for specifying business transactions and includes compensation constructs. We show how a substantial subset of BPEL can be mapped to StAC thus demonstrating the expressiveness of StAC and providing a formal semantics for BPEL.

Key Words: Long-running transactions, compensation, language semantics, formal specification, B Method.
Category: D.3.1, F.3.2, H.m

1 Introduction

Business transactions involve hierarchies of activities whose execution needs to be orchestrated. Business transactions typically involve interactions and coordination between multiple partners. The behavior of a transaction can depend on data passed between partners and can involve the manipulation of data. Business transactions also need to deal with faults that arise at any stage of execution. In standard atomic transactions, such as database transactions, rollback mechanisms are used to protect against faults by providing all or nothing atomicity for transactions [16]. In so-called long running business transactions 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. In such cases compensation can be used as a way of dealing with faults.

In the context of business transactions, Gray [16] defines a compensation as the action taken to recover from error or cope with a change of plan. 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. Garcia-Molina and Salem [15] used compensation to define the concept of *sagas*. A saga partitions a long running transaction into a sequence of several smaller subtransactions, where each of the subtransactions has an associated compensation. If one of the subtransactions in the sequence aborts, the compensation associated with those committed subtransactions is executed in reverse order.

In terms of the range of techniques used for system dependability [2], our approach to compensation may be viewed as a forward error recovery mechanism. For example, when a bookstore realises that a book that has been ordered by a customer is out of print so is unable to deliver it (the error state), it recovers from this error by performing various actions (offering a refund and sending an apology). It does not recover by rolling back to some previous state. The saga approach [15] requires that compensations represent a semantic undo of the forward action in the usual database sense and so their approach may be regarded as a form of backwards error recovery. Although our approach does not require that compensations perform a semantic undo, it is fair to say that the coordination of compensations has been influenced by the requirement for semantic undo, i.e., in the case that compensations perform a semantic undo, then invocation of compensations can be regarded as a backward error recovery mechanism.

The StAC language (**St**ructured **A**ctivity **C**ompensation) was introduced in [6] as a business process modelling language and includes constructs for modelling compensation in business processes. StAC is based on process algebras such as Milner's Calculus for Communicating Systems (CCS) [20] and Hoare's Communicating Sequential Processes (CSP) [17]. 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. StAC gives a precise interpretation to the mechanics of compensation, including the combination of compensation with parallel execution, hierarchy and exceptions.

To facilitate modelling of data dependency and manipulation, a StAC process has global data state associated with it and this state is changed by atomic actions. Typically the data state of a StAC process is represented using state variables and the effect of atomic activities on data is represented using assignment to variables. Rather than inventing our own notation for specifying data variables and operations for manipulating them, we have chosen to use the B notation of Abrial's B Method [1]. B provides a state based formal notation for writing precise abstract specifications of data and data manipulation and makes strong use of set theory and predicate logic. In our approach a business transaction specification has two components: the StAC specification that describes the orchestration of activities, including ordering of compensation activities, and a B specification that describes an abstract model of the data of the transaction along with atomic actions for manipulating the data.

The combination of StAC and B provides a clear and succinct notation for writing precise specifications of long running business transactions. The notation has a formal semantics ensuring specifications are unambiguous and amenable to formal analysis. When business transactions are provided in the open web services environment [3], we believe that the ability to write precise specifications is particularly important. As well as providing the basis for rigorous implementation of business transactions, precise specifications can act as contracts for business services. Such contracts may be shared with partners to provide them with sufficient detail to be able to interact sensibly with a service, without having access to all implementation details.

StAC was originally inspired by the BPBeans development framework [11] that allows an application to be built by a nested hierarchy of business processes. BPBeans supports orchestration of activities through parallel and sequential composition, as well as exception and compensation handling. Similar orchestration mechanisms are also found in the BizTalk [19] and BPEL4WS [12] languages. BPEL4WS (Business Process Execution Language for Web Services), or BPEL for short, is a web services composition language that is jointly developed by Microsoft, IBM and others. It provides an XML syntax for writing abstract and/or executable specifications of business processes which may be provided as web services and may involve the invocation of other web services.

In this paper, as well as providing an overview of how StAC and its combination with B may be used to write precise specifications of business transactions, we show how the notations may be used to give a formal semantics to the BPEL language. We take a substantial subset of the BPEL language and define a mapping from that language subset to StAC. This mapping illstrates the expressiveness of StAC and we also believe it makes BPEL semantics more

P	::=	$skip$	(successful termination)
	$\mid$	N	(call named process)
	$\mid$	A	(atomic action label)
	$\mid$	$b \Longrightarrow P$	(conditional process)
	$\mid$	$P;Q$	(sequencing)
	$\mid$	$P \underset{X}{\parallel} Q$	(parallel)
	$\mid$	$P \setminus S$	(event hiding)
	$\mid$	$P \Box Q$	(external choice)
	$\mid$	$P \sqcap Q$	(internal choice)
	$\mid$	$P\{Q\}R$	(attempt block)
	$\mid$	$\odot$	(abnormal termination)
	$\mid$	$P \div Q$	(compensation pair)
	$\mid$	$\boxtimes$	(reverse)
	$\mid$	☑	(accept)
	$\mid$	$[P]$	(compensation scoping)

Table 1: StAC Syntax

accessible through a succinct but precise formal definition.

Section 2 introduces the StAC language and illustrates its expressiveness through an example. Section 3 describes how StAC may be combined with the B notation. Section 4 shows how StAC may be used to give a semantics to BPEL activity constructs. Section 4 only supports a certain form of BPEL compensation. In Section 5 the StAC language is extended in a way that allows BPEL compensation to be modelled more fully.

2 The StAC Language

The StAC language provides standard combinators for modelling sequential and parallel processes, along with specific combinators to deal with compensation. The syntax of StAC processes is presented in Table 1.

A process is specified 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 process *skip* does nothing and immediately terminates successfully. For process name N, the process N calls the process named N returning if and when N terminates successfully. Mutually recursive calls are possible.

Another basic process is one that performs an atomic action labelled A. Each activity label A has an associated action $\overset{A}{\to}$ representing an atomic change in the data state: if Σ is the set of all possible data states, then $\overset{A}{\to}$ is a relation on

Σ. In Section 3 we will show how B provides a rich notation for specifying data states and atomic actions on data states. In this section we focus on the StAC operators that provide the orchestration mechanisms for long running business transactions.

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

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 $P \parallel Q$.

With hiding one can make the execution of activities invisible to the environment: $P \setminus C$ represents the process P with all the events in the set C hidden from the environment.

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:

$$[]\, b \in BOOK \bullet ChooseBook(b)$$

Details of the generalised versions of the operators may be found in [13]

2.1 Attempt Block and Early Termination

An important feature 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 call 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 $skip\{Q\}skip$.)

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.

2.2 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 process, and once the primary process has successfully completed, the compensation process is remembered (installed) for possible later invocation.

We refer to the invocation of compensation activities as *reversal*. If we reach a point where compensation will no longer be required, compensation activities can be forgotten. We refer to this as *acceptance*. 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 $A; A'$.

The *accept* operation ($\checkmark\!\!\!\!\square$) indicates that currently installed compensations should be cleared, meaning that after an accept the compensation task is set to *skip*. The process $(A \div A'); \checkmark\!\!\!\!\square; \boxtimes$ executes A and when the $\boxtimes$ operation is called the compensation task A' has already been cleared by the $\checkmark\!\!\!\!\square$ operator so A' will not be executed.

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 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 $A' \parallel B'$.

Next we will consider the combination of compensation with choice. The process $(A \div A') \,[]\, (B \div B')$ behaves either as A or as B with the choice between A and B being 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 will 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 $(A \div B)$.

2.3 Scoping of Compensation

The compensation scoping brackets $[\cdots]$ provide nested compensation scoping and are used to delimit the scope of the acceptance and reversal operators. All StAC processes have an implicit outer compensation scope. The start of a 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 it is 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 available 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.

2.4 Example: Order Fulfillment

To illustrate the use of StAC we present the order fulfillment example described in [9] and [10]. 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:

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

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

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

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 simply adds 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.

The bold font, e.g., **AcceptOrder**, indicates an atomic action. In Section 3 the effect of these and other actions on a data state will be specified using the B notation.

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:

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

Here **okCreditCheck** is a boolean expression that will be specified in terms of a data state in Section 3. 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:

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

This 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. The *PackOrder* process packs each of the items in the order in parallel. Each **PackItem** activity is reversed by a corresponding **UnpackItem**:

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

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. Because the **AcceptOrder** $\div$ **RestockOrder** pair is composed sequentially with the $FulfillOrder$ process, **RestockOrder** will not happen until all appropriate compensations of $FulfillOrder$ have been completed.

2.5 Overview of StAC Formal Semantics

We do not present the formal semantics of StAC here since they already appear in [7, 13]. Instead we briefly provide an overview of the nature of the formal semantics. [7, 13] use a so-called Plotkin style of operational semantics [22] to define StAC where a set of transition rules are used to define transitions between configurations. For the operational semantics of StAC, a configuration is a tuple:

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

In the above tuple, C is a function that returns the compensation process $C(i)$, for each compensation index i. Scoping of compensation is modelled using compensation indices, with each scope having its own index. 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. In Section 5 we will show how compensation indices may be introduced explicitly into the language to provide a multiple compensation facility.

A process gives rise to a labelled transition system with configurations being the nodes of the system and activity labels being the transition labels. The labelled transition

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

denotes that the execution of a basic activity A may cause a configuration transition from $(P,\ C,\ \sigma)$ to $(P',\ C',\ \sigma')$.

A set of transition rules define the conditions under which labelled transitions may occur. An atomic action is a relation from Σ to Σ, and we write $\sigma \overset{A}{\to} \sigma'$ when σ is related to σ' by $\overset{A}{\to}$. The following transition rule shows that execution of an atomic action leads to successful termination (skip) with a new data state σ' and an unchanged compensation function:

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

This rule should be read as: if σ is related to σ' by $\overset{A}{\to}$, then there is an A-labelled transition from process configuration $(A,\ C,\ \sigma)$ to configuration $(\mathit{skip},\ C,\ \sigma')$.

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

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

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

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

The label τ is a special label that represents an operation not visible to the external environment.

As mentioned above, each compensation scope is given an explicit compensation index. To define the semantics of compensation, each occurrence of a compensation operator in a term is decorated with the index corresponding to the scope in which it appears. The following rule for the compensation pair operator adds the compensation process Q to the compensation function C when the primary process has terminated sucessfully:

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

$C[\; i := (Q; C(i))\;]$ 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.

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

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

The full set of transition rules for StAC may be found in [7].

3 Describing State and Activities in B

The B notation is a state based formal notation that is part of the B method developed by Abrial [1]. In the B method, a system is defined as an abstract machine consisting of some state and some operations acting on the state. An abstract machine has the structure presented in Figure 1. The abstract machine M consists of some variables V, an invariant I and some atomic actions on the variables. Operations act on the variables while preserving the invariant and can have input and output parameters. The given sets S declare some basic types that may be used to type the variables. The type constructors of B are standard constructors of set theory such as powersets, relations and functions. These set theoretic constructors allow for models of the states of a system that abstract from implementation detail.

Initialisation and operations are written in the generalised substitution notation of B, which includes constructs such as assignment, guarded statements, and choice. In the assignment statement $x := E$, x is a variable and E is an expression that may use any of the available variables. Simultaneous assignment $x := E \parallel y := F$ is equivalent to $x, y := E, F$. In the guarded statement

$$G \implies S$$

MACHINE M
SETS S
VARIABLES V
INVARIANT I
INITIALISATION $init$
OPERATIONS
$\quad \ldots$
END

Figure 1: B abstract machine

the guard G is a condition on the state variables and S is a generalised substitution. This statement will be enabled only when G holds. The nondeterministic choice between two statements is written

$$\textbf{CHOICE}\ \ S\ \ \textbf{OR}\ \ T\ \ \textbf{END}$$

The choice is enabled when either S or T is enabled. The unbounded choice

$$\textbf{ANY}\ \ x\ \ \textbf{WHERE}\ \ P\ \ \textbf{THEN}\ \ S\ \ \textbf{END}$$

nondeterministically chooses some value x satisfying P and then behaves like S. The ANY statement has an implicit guard: it is only enabled if there is some x satisfying P.

A business transaction may be specified as a combination of a B machine and a StAC process. The B machine defines a data state and atomic actions on that state. The StAC part orchestrates the order in which actions are executed and has access to the data state in defining conditional behaviour. Guarded atomic actions can further constrain the orchestration. For example, if A is an atomic event of the form $G \Longrightarrow S$, then the StAC process $A; P$ will only be enabled if G holds.

Recall from Section 2.5 that we assume a data space Σ and a set of labelled relations on Σ. When the data is defined by a B machine, the cartesian product of the types of each of the machine variables represents the space Σ. The operations of the machine define relations on the state space. Details of this may be found in [14]. Our approach to combining StAC and B is similar to the approach taken in [5] to combining CSP and B.

```
MACHINE  $ACME$
SETS
    $ITEM$
VARIABLES
    $stock, order, courierBooking, creditStatus, packaging$
DEFINITIONS
    $inStock(s) \ == \ \forall i \cdot (i \in s \Rightarrow stock(i) > 0)$
INVARIANT
    $stock \ \in \ ITEM \rightarrow \mathbb{N} \ \wedge$
    $order \ \subseteq \ ITEM \ \wedge$
    $courierBooking \ \in \ \mathbf{BOOL} \ \wedge$
    $creditStatus \ \in \ \mathbf{BOOL} \ \wedge$
    $packaging \ \subseteq \ order$
```

Figure 2: ACME machine state

3.1 Order fulfillment example

The abstract machine for the $ACME$ transaction is shown in Figure 2. This has a single set *ITEM* that represents all items available to the customers. The VARIABLES clause declares the variables of the abstract machine such as *stock*, *order*, *courierBooking*, *okCreditCheck*, and *packaging*. In the INVARIANT we specify the types of the variables introduced in the previous clause. Variable *stock* is a total function abstractly modelling the stock inventory. The *stock* variable maps each item identifier to the remaining quantity of that item. Variable *order* represents the items chosen by the client, and is defined as a subset of *ITEM*. The variables *courierBooking* and *okCreditCheck* are boolean variables: *courierBooking* says whether or not the courier was booked; *okCreditCheck* represents the outcome of the credit check. The last variable, *packaging*, is a subset of *order* that contains those items in the order already packed. In the DEFINITIONS clause we have defined the boolean expression *inStock* that checks whether all the items ordered by the client are available.

The machine operations define the atomic actions used in the $ACME$ process. The **AcceptOrder** action uses the ANY construct to nondeterministically choose a set of items that satisfy the condition *inStock*, assuring that all the items chosen are available in stock. Three simultaneous substitutions are specified using the selected set of items: the stock for each of the selected items is decreased by one; the set *items* is assigned to the customer order; and the packaging of the items in the customer order is initialised to the empty set:

```
**AcceptOrder** $\hat{=}$
  **ANY** $items$ **WHERE** $items \subseteq ITEM \ \wedge \ inStock(items)$ **THEN**
    $stock := stock \lhd \lambda\,(item)\,.\,(item \in items \mid stock(item) - 1) \ \ \|$
    $order := items \ \ \|$
    $packaging := \{\}$
  **END**
```

The expression $f \lhd \lambda(x).(x \in X \mid h(x))$ describes a multiple update of a function f such that each $x \in X$ is mapped to a new value $h(x)$.

In the **RestockOrder** action, provided the packaging is empty, the stock for all items in the customer order is increased by one, making those items available to other customers. Simultaneously with the stock update, the customer order is emptied:

```
**RestockOrder** $\hat{=}$
  $packaging = \{\} \implies$
    $stock := stock \lhd \lambda\,(item)\,.\,(item \in order \mid stock(item) + 1) \ \ \|$
    $order := \{\}$
```

The **CreditCheck** action sets the variable **okCreditCheck**. This is used by the *FulfillOrder* process to trigger an early termination when the client's credit card is rejected:

```
**CreditCheck** $\hat{=}$
  **BEGIN**
    **CHOICE**
      $okCreditCheck := \text{true}$
    **OR**
      $okCreditCheck := \text{false}$
    **END**
  **END**
```

The **CreditCheck** action is described as a choice between assigning the value *true* or *false* to the variable **okCreditCheck** depending on the credit card being accepted or rejected. This a simple abstraction of the real processing which may involve getting authorisation from a credit card company.

The **BookCourier** action represents the booking of the courier, which is done by setting the variable *courierBooking* to *true*. The **CancelCourier** action is similar except that it sets *courierBooking* to *false*:

```
**BookCourier** $\hat{=}$
    $courierBooking := \text{true}$
```

$$
\begin{array}{l}
\textbf{CancelCourier} \mathrel{\hat{=}} \\
\quad \mathit{courierBooking} := \text{false}
\end{array}
$$

The **PackItem** action adds an item to the set *packaging* and removes it from the order, while **UnpackItem** reverses this:

$$
\begin{array}{l}
\textbf{PackItem(i)} \mathrel{\hat{=}} \\
\quad i \in \mathit{order} \implies \\
\qquad \mathit{packaging} := \mathit{packaging} \cup \{i\} \parallel \\
\qquad \mathit{order} := \mathit{order} \setminus \{i\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{UnPackItem(i)} \mathrel{\hat{=}} \\
\quad i \in \mathit{packaging} \implies \\
\qquad \mathit{order} := \mathit{order} \cup \{i\} \parallel \\
\qquad \mathit{packaging} := \mathit{packaging} \setminus \{i\}
\end{array}
$$

4 Formalising BPEL process behaviour

4.1 Overview of BPEL

A BPEL process performs hierarchically orchestrated activities including receiving messages from other services and sending messages to other services. The other services which interact with a BPEL process are referred to as partners and the process communicates messages with its partners through ports. The BPEL language provides constructs for declaring message types, port types, partner types, and process variables for storing messages. It also provides constructs for defining basic activities and structured activities. Examples of basic activities include message receipt and sending, variable assignment, and throwing an exception. Examples of structured activities include sequencing, parallel flows and conditional branching.

The main activity structuring mechanism for declaring hierarchical units of work is the BPEL scope. A BPEL scope is partitioned into four sections (Figure 3):

1. The *main body* which contains the description of the normal behaviour. The main body can consist of any basic or structured activities (including scopes).

2. The *event handlers* which will are invoked when certain events occur. An event might be the receipt of a particular type of message from a partner or a timeout.

3. The *compensation handler* contains activities which are intended to compensate the effect of the main body. A compensation handler is installed on successful completion of the main body for possible later execution.

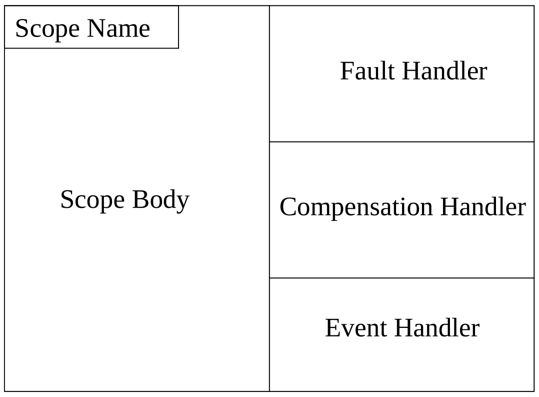

Figure 3: BPEL Scope.

4. The *fault handlers* catch faults that are thrown by the main body or by one of the event handlers.

Previously installed compensation handlers are invoked through the BPEL *compensate* activity. The compensate activity can only appear in a fault handler or a compensation handler, that is, a fault or compensation handler for a scope S may invoke compensations on scopes directly within the main body of S. The compensate activity comes in two forms. In the first form (the *named* form), a scope name is provided which means that the installed compensation handler for the scope identified by the argument should be invoked. In the second form (the *name-free* form), no scope argument is provided and the default compensation is applied. Default compensation means that the installed compensation handlers for all the direct inner scopes will the invoked. In this section we will only model the name-free form of compensation invocation. In section 5 we will describe an extension to StAC and we will show how this allows us to model the named form of explicit compensation.

In this paper we focus on formalising the basic and structured activity constructs of BPEL. We will not be concerned with formalising BPEL message, partner nor variable types. We will see that all the significant activity constructs of BPEL can be mapped easily to StAC. Before we present the mapping from BPEL activities to StAC, we describe a modest extension to StAC which supports throwing and handling of different types of exception. This extension is necessary to model BPEL fault handling.

4.2 Extending StAC with Parameterised Throw

Although StAC deals with error handling in a similar way to BPEL, StAC does not distinguish between different faults as BPEL does. This section discusses how StAC could be extended to deal with parameterised throw and catch of exceptions so that early termination attempt and attempt block constructs can distinguish the type of exception raised and execute an exception handler tailored for each exception. Constructs that deal with exception handling are parameterised by the exception identifier:

$$
\begin{array}{lll}
\mathrm{P} ::= \cdots & & \\
\quad \mid \odot_f & & \text{(early termination of attempt)} \\
\quad \mid P\{Q\}(\ \square e \in E \cdot\ e \Longrightarrow R_e\ ) & & \text{(attempt block)} \\
\quad \mid \cdots & &
\end{array}
$$

where E is a set of exception identifiers and f is an exception identifier. Furthermore, the attempt block now provides a choice of exception handlers instead of a single handler.

An extended attempt block $P\{Q\}(\ \square e \in E \cdot\ e \Longrightarrow R_e\ )$ first executes Q, and if Q terminates successfully it continues with P. Otherwise, if an early termination $\odot_f$ occurs within Q, there are two possible outcomes. Either the exception f has an exception handler R_f (with $f \in E$), in which case the block continues with R_f, or f does not have an exception handler ($f \notin E$), in which case the exception $\odot_f$ is thrown again to the surrounding block. Appendix A shows how the semantic rules for StAC in [7] may be extended to deal with parameterised throw.

4.3 Mapping BPEL to StAC

To define a mapping from BPEL to StAC, we have defined an abstract syntax for a simplified subset of the activity description language of BPEL. This abstract syntax for BPEL activities is shown in Table 2. We have assumed some basic sets such as $Partner$, representing a set of identifiers for partners, and $Operation$, representing a set of operation identifiers. In the remainder of this section we will go through each syntactic construct in Table 2 and explain how it is mapped to StAC. We introduce a function $\mathbb{T}$ that maps each syntactic construct of BPEL to an appropriate corresponding StAC construction. For simplicity we are not modelling BPEL process variables nor assignment to variables in this paper since we want to focus on formalising the orchestration of activities in BPEL. Another important feature of BPEL that we do model in this paper is the creation of process instances. In BPEL a receive activity can be defined so that receipt of a message creates a new process instance. Correlation rules can be defined which correlate messages with the appropriate instance of a process. We do not treat correlation sets or time related features in our formalisation of BPEL.

The *receive* activity allows a process to receive a message from a partner. The receive operation specifies which partner it expects to receive the message from and the operation it expects the partner to invoke. Thus, a receive is of the form $receive(p, op)$, where $p \in Partner$, $op \in Operation$. A BPEL *receive*

$Act ::= receive(Partner, Operation)$	(receive a message)
$\mid\ reply(Partner, Operation)$	(reply to a message)
$\mid\ invokeS(Partner, Operation)$	(synchronous operation invoke)
$\mid\ invokeA(Partner, Operation)$	(asynchronous operation invoke)
$\mid\ sequence(Act_1, \ldots, Act_n)$	(sequence)
$\mid\ while(Cond, Act)$	(conditional loop)
$\mid\ switch(CAct_1, \ldots, CAct_n, Act)$	(choice of conditional activities)
$\mid\ pick(GAct_1, \ldots, GAct_n)$	(choice of guarded activities)
$\mid\ flow(Act_1, \ldots, Act_n)$	(parallel flow)
$\mid\ linkedActivity(Links_{in}, Act, Links_{out})$	(activity with incoming and outgoing links)
$\mid\ scope(ScopeName, Act, EH, FH, CH)$	(scope)
$\mid\ throw(FaultName)$	(throw a fault)
$\mid\ compensate(ScopeName)$	(compensate a scope)

$$\begin{aligned}
CAct &::= conditionalActivity(Cond, Act)\\
GAct &::= guardedActivity(receive(Partner, Operation, Var_{in}), Act)\\
EH &::= eventHandler(GAct_1, \ldots, GAct_n)\\
FH &::= faultHandler(\ catch(FaultName_1, Act_1), \ldots,\\
&\qquad\qquad catch(FaultName_n, Act_n)\ )\\
CH &::= compensationHandler(Act)
\end{aligned}$$

Table 2: Abstract syntax for BPEL subset

is modelled simply as an atomic event in StAC, thus the results of translating a *receive* is defined simply as follows:

$$\mathbb{T}(\ receive(p, op)\ )\ =\ receive.p.op$$

A *reply* specifies a partner and an operation with which the reply is concerned:

$$\mathbb{T}(\ reply(p, op)\ )\ =\ reply.p.op$$

Invocation is used by one BPEL process to request an operation on another BPEL process. Invocation comes in two forms, asynchronous, in which the requester does not wait for a reply, and synchronous, in which the requester does wait for a reply. In the asynchronous case, the invoked process replies to the original request by invoking the original requestor. A synchronous *invokeS* activity consists of two StAC activities, sending a request to a partner followed by getting a response from the partner:

$$\mathbb{T}(\ invokeS(p, op)\ )\ =\ request.p.op\ ;\ response.p.op$$

Although not defined in this paper, our assumption is that a $\mathit{request}.p2.\mathit{op}$ event initiated by partner $p1$ would result in a corresponding $\mathit{receive}.p1.\mathit{op}$ event occurring in partner $p2$. Similarly for $\mathit{reply}.p1.\mathit{op}$ and $\mathit{response}.p2.\mathit{op}$. These assumptions represent the way in which we intend BPEL processes to interact. An asynchronous $\mathit{invokeA}$ activity simply involves sending a request to a partner:

$$\mathbb{T}(\ \mathit{invokeA}(p, \mathit{op})\ ) \ = \ \mathit{request}.p.\mathit{op}$$

Given a group of activities $A_1, \ldots, A_n$, the structured activity $\mathit{sequence}(A_1, \ldots, A_n)$ represents the execution of these activities in sequential order. Sequential composition of activities is defined in terms of the sequential composition operator of StAC:

$$\mathbb{T}(\ \mathit{sequence}(A_1, \ldots, A_n)\ ) \ = \ \mathbb{T}(A_1)\ ;\ \cdots\ ;\ \mathbb{T}(A_n)$$

Looping behaviour is defined in BPEL using the *while* construct. This has the usual semantics for while: $\mathit{while}(C, A)$ continues to execute A while condition C holds. For simplicity we assume that conditions are expressed in the same language as StAC boolean conditions. The *while* construct is defined in terms of a recursive StAC definition:

$$\begin{array}{l} \mathbb{T}(\ \mathit{while}(C, A)\ ) \ = \ W \\ \textbf{where} \\ \quad W \ = \ C \Longrightarrow (\mathbb{T}(A)\ ; W) \ \ [] \ \ \neg C \Longrightarrow \mathit{skip} \end{array}$$

Here process W will execute $\mathbb{T}(A)$ and then continue as W provided condition C holds, otherwise it terminates immediately.

The BPEL *switch* selects one of a group of activities for execution. Each activity has an associated selection condition. Conditions are evaluated in order of appearance and the first branch whose condition is true is evaluated. If no activity satisfies its selection condition, the default activity is executed. The *switch* is translated to StAC as follows:

$$\begin{array}{l} \mathbb{T}(\ \mathit{switch}(\mathit{conditionalActivity}(C_1, A_1), \ldots, \mathit{conditionalActivity}(C_n, A_n), A)\ ) \\ \quad = \ C_1 \ \Longrightarrow \ \mathbb{T}(A_1) \ \ [] \\ \qquad \neg C_1 \wedge C_2 \ \Longrightarrow \ \mathbb{T}(A_1) \ \ [] \\ \qquad \ldots \\ \qquad \neg C_1 \wedge \cdots \wedge \neg C_{n-1} \wedge C_n \ \Longrightarrow \ \mathbb{T}(A_n) \ \ [] \\ \qquad \neg C_1 \wedge \cdots \wedge \neg C_{n-1} \wedge \neg C_n \ \Longrightarrow \ \mathbb{T}(A) \end{array}$$

The BPEL *pick* is similar to the *switch* except that instead of being guarded by a condition, each activity is guarded by a message receipt event. The *pick* waits until it receives a message matching one of the activity guards. When it receives a matching message, it executes the corresponding activity. The *pick*

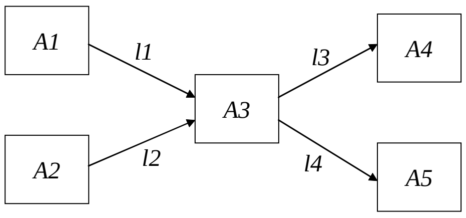

Figure 4: Example BPEL flow with links.

corresponds to the StAC choice, where each branch is guarded by an event, and is translated to StAC as follows:

$$\begin{aligned}&\mathbb{T}(\ \mathit{pick}(\mathit{guardedActivity}(R_1, A_1), \ldots, \mathit{guardedActivity}(R_n, A_n))\ )\\ &\quad = \ \mathbb{T}(R_1); \mathbb{T}(A_1) \quad [] \ \cdots \ [] \quad \mathbb{T}(R_n); \mathbb{T}(A_n)\end{aligned}$$

The BPEL syntax allows multiple simultaneous receives on the same message type from the same partner though the meaning of such behaviour is said to be undefined in [12]. StAC supports multiple simultaneous receives though it can lead to nondeterministic choice of behaviour.

The BPEL $flow$ construct is used to specify a group of activities running in parallel. Synchronisation dependencies between activities may be specified in BPEL flows using dependency links. Each link has exactly one source and one target. For example, consider the dependent flow illustrated in Figure 4. This flow contains five parallel activities. The arrows represent synchronisation dependencies. The arrow from $A1$ to $A3$ indicates that activity $A3$ cannot commence until $A1$ has completed. When an activity has multiple incoming links, then it must wait until all the source activities of its incoming links have completed. For example, $A3$ in Figure 4 must wait until both $A1$ and $A2$ complete. In our abstract syntax for BPEL we can attach incoming and outgoing links to an activity using the *linkedActivity* construct. If $L1$ and $L2$ are sets of links, then $\mathit{linkedActivity}(L1, A, L2)$ represents an activity A with incoming links $L1$ and outgoing links $L2$. The example of Figure 4 would be represented in our BPEL syntax as:

$$\begin{array}{l} flow(\\ \quad \mathit{linkedActivity}(\{\}, A1, \{l1\}),\\ \quad \mathit{linkedActivity}(\{\}, A2, \{l2\}),\\ \quad \mathit{linkedActivity}(\{l1, l2\}, A3, \{l3, l4\}),\\ \quad \mathit{linkedActivity}(\{l3\}, A4, \{\}),\\ \quad \mathit{linkedActivity}(\{l4\}, A5, \{\})\\ ) \end{array}$$

We assume that links do not cross the boundaries of a flow nor the boundaries of a while-loop.

We model synchronisation links in StAC by using synchronisation variables shared between processes. Each link has a boolean variable associated with it. Initially the variable is assumed to be $false$. When the activity at the source of a link completes, it sets the link variable to $true$. A linked activity must wait until all its incoming links are $true$ before proceeding. An atomic action which sets a set of links to $true$ is defined in B as follows[1]:

$$Set(L) \quad = \quad (\|_{l \in L} \cdot l := true)$$

Waiting for all incoming links to become $true$ is modelled using a guarded atomic action defined in B as follows:

$$Wait(L) \quad = \quad (\forall\, l \in L \cdot l = true) \implies skip$$

Thus, for example, the linked activity containing $A3$ in our example is translated to StAC as:

$$Wait(\{l1, l2\}) \ \ ; \ \ \mathbb{T}(A3) \ \ ; \ \ Set(\{l3, l4\})$$

When this process is run in parallel with the appropriate translations of the other linked activities of our example, $A3$ cannot occur until $A1$ and $A2$ have completed and have set $l1$ and $l2$ to $true$.

A flow is translated into a parallel composition in StAC:

$$\mathbb{T}(\ flow(A_1, \ldots, A_n)\ ) \quad = \quad (\|_{l \in L} \cdot l := false) \ \ ; \ \ (\ \mathbb{T}(A_1) \ \| \ \cdots \ \| \ \mathbb{T}(A_n)\ )$$

Here the links used in the flow are initialised to false. Although not defined formally here, the set L represents the set of link identifiers contained in the flow excluding those contained in nested flows or loops.

To achieve the desired link dependency, the translation rule for linked activities uses the $Wait$ and Set actions as follows:

$$\mathbb{T}(\ linkedActivity(L1, A, L2)\ ) \quad = \quad Wait(L1) \ \ ; \ \ \mathbb{T}(A) \ \ ; \ \ Set(L2)$$

BPEL link sources may have transition conditions and linked activities may have join conditions which are expressions of the incoming transition conditions. We do not model link conditions in this paper which represents a big simplification of the BPEL link semantics.

We now consider how to model the BPEL scope construct. Recall that as well as a main body, a BPEL scope has an event handler, a fault handler and a compensation handler. We model compensation handling using the StAC compensation pair operator and fault handling using the StAC attempt block. We

[1] Here we are taking some syntactic libertes with the B notation. A valid B representation would be to model the link variables a a function $l \in Link \rightarrow Bool$ and define $Set(L)$ using function update.

model the event handler as a process that runs in parallel with the main body. A scope of the form

$$scope(S, A, EH, FH, CH)$$

will be modelled with the following StAC construction:

$$[\ \ skip\ \{\ (A \parallel EH) \div CH\ \}\ FH\ \ ]$$

This is a compensation scope containing an attempt block. The main body of the attempt block consists of the scope body A in parallel with the event handler EH. If $A \parallel EH$ terminates normally, the compensation handler CH is installed and the whole scope terminates. A fault may be thrown by A or by EH in which case control is passed to the fault handler FH and CH is not installed.

Each event handler is guarded by a receive event and can be executed multiple times. An event handler is disabled when the main body terminates (normally or through a fault). We model this in StAC using a looping construct that on each iteration is either willing to terminate immediately or to execute process P. This is defined as follows:

$$\begin{array}{l} LOOP(P) \ = \ X \\ \textbf{where} \\ \quad X \ = \ skip \ [] \ (P; X) \end{array}$$

A scope may contain a group of event handlers for different messages and these should be available in parallel:

$$eventHandler(guardedActivity(R_1, A_1), \ldots, guardedActivity(R_n, A_n))$$

Here each R_i is a receive activity which is bocked until an appropriate message is received in which case the corresponding A_i activity is executed. The event handlers are modelled in StAC by looping over the handlers in parallel:

$$LOOP(R_1; A_1) \ \ \parallel \ \cdots \ \parallel \ \ LOOP(R_n; A_n)$$

As well as message receipt, BPEL also allows event handlers to be guarded by timeout alarms. We have not modelled alarms here.

A BPEL fault handler is of the form[2]

$$faultHandler(catch(f_1, F_1), \ldots, catch(f_n, F_n))$$

Here f_i is a fault type and A_i is the corresponding handler for f_i. This will be modelled in StAC using the parameterised exception operators introduced in Section 4.2.

[2] We do not deal with the *catchall* form of handler.

When the compensation handler for a BPEL scope is installed it overrides any compensation handlers installed by nested inner scopes. We use the $StAC$ $\boxtimes$ operator to clear inner compensations when installing the outer compensation. $[P; \boxtimes] \div Q$ will clear any compensations installed by P before installing Q. If P throws an exception, then the compensations installed by P will remain available and Q will not be installed. We believe this corresponds to the informal BPEL semantics defined in [12] in the case that the name-free form of compensation is used.

The rule for translating a BPEL scope is as follows:

$$\mathbb{T}(\ scope(S, A, EH, FH, CH)\ ) \ = \ [\ \ skip\ \{\ [\ (A' \parallel EH'); \boxtimes\ ] \div CH'\ \}\ FH'\ \ ]$$

where

$$\begin{aligned} A' &= \mathbb{T}(A) \\ EH &= eventHandler(\,guardedActivity(R_1, A_1), \ldots, \\ &\qquad\qquad\qquad guardedActivity(R_n, A_n)\ \ ) \\ EH' &= LOOP(\mathbb{T}(R_1); \mathbb{T}(A_1)) \ \parallel \ \cdots \ \parallel \ LOOP(\mathbb{T}(R_n); \mathbb{T}(A_n)) \\ FH &= faultHandler(catch(f_1, F_1), \ldots, catch(f_n, F_n)) \\ FH' &= f_1 \Longrightarrow \mathbb{T}(F_1) \ \ [] \ \ldots \ [] \ \ f_n \Longrightarrow \mathbb{T}(F_n) \\ CH &= compensationHandler(C) \\ CH' &= \mathbb{T}(C) \end{aligned}$$

The compensation handler is optional in StAC. In the case that a compensation handler is not present, the default compensation for a scope is to run the compensations of the enclosed scopes. To model this in StAC, inner compensations are not cleared and no compensation handler is installed:

$$\mathbb{T}(\ scope(S, A, EH, FH, \_\,)\ ) \ = \ [\ \ skip\ \{\ A' \parallel EH'\ \}\ FH'\ \ ]$$

where A', EH' and FH' are as defined above.

The fault handler is also optional in BPEL. If it is not present, the default is to invoke the default compensation for the scope and throw the exception to the outer level. If the fault handler is not present, the above definition of FH' should be replaced by the following:

$$FH' \ = \ []e \in \mathcal{E} \cdot \ e \ \Longrightarrow \ \boxtimes; \odot_e$$

Here $\mathcal{E}$ represents the set of all possible exceptions types. The above construction defines a handler for every exception type, the effect of which is to invoke compensation and then re-throw that exception.

The event handler is also optional in BPEL. When it is absent, EH' is simply *skip*.

A BPEL throw activity may appear anywhere in a scope. The BPEL throw activity is modelled in StAC as follows:

$$\mathbb{T}(throw(f)) = \odot_f$$

The BPEL compensate activity can only appear in a compensation handler or a fault handler. As mentioned at the beginning of the section, for the moment we provide a translation for the name-free compensate activity. The name-free compensate activity causes all the immediately enclosed scopes to be compensated, similar to the default case where no handler is present. The name-free compensate is translated as follows:

$$\mathbb{T}(compensate) = \boxtimes$$

The BPEL standard [12] says that when the default compensation is applied to a scope, the compensation handlers should be be run in reverse order of completion of the scopes to which they are attached. Because of the semantics of StAC, we give a less constrained semantics to default compensation, that is, compensations for sequentially composed scopes are executed in reverse order while compensations for scopes composed in parallel can be run in parallel. Though our interpretation allows for more parallelism when compensating, it may not be appropriate in the presence of link dependencies in flows.

In the next section we introduce a modified form of the StAC language that supports multiple compensation threads. This will allow us to model the named compensate activity.

5 Multiple Compensation in StAC

In this section we present some extensions to the StAC language. The most important of these extensions is that 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 3.

The original motivation for extending the StAC language was that StAC$_i$ had a clear semantics for compensation and made it easier to describe parallel compensation. It is easier to define the operational semantics of multiple compensation tasks, than a hierarchy of compensation scopes, because with multiple compensation it is possible to refer directly to a compensation task by its index, while with nested compensations this is not possible. Later, when applying StAC to some case studies it emerged that some features that were difficult to model in StAC could be easily modeled in StAC$_i$ using multiple compensation tasks. This suggests that multiple compensation is a useful concept.

$$\begin{array}{rcll} \text{P,Q} & ::= & \cdots & \\ & | & P \div_i Q & \text{(indexed compensation pair)} \\ & | & \boxtimes_i & \text{(indexed reverse)} \\ & | & \boxdot_i & \text{(indexed accept)} \\ & | & J \rhd i & \text{(merge)} \end{array}$$

Table 3: StAC$_i$ Syntax

5.1 Extended Compensation Operators

Most of the StAC$_i$ operators are retained from StAC without any alterations. The new operators deal with compensation (Table 3) and reflect the modifications to the StAC language. These replace the compensation operators of StAC presented in Section 2. In the extended language, process $P \div_i Q$ has P as its primary process and, when P completes, compensation Q is installed on compensation task i, where i is an index. Note that compensation indices are constants and not expressions that can be evaluated. The instruction to accept compensation task i is given by $\boxdot_i$ while the instruction to reverse compensation task i is given by $\boxtimes_i$. To help illustrate indexed compensation, consider the process:

$$(A \div_1 A'); (B \div_2 B'); \boxtimes_1; (C \div_2 C'); \boxtimes_2.$$

This process will start by invoking A, followed by B and then the reversal causes compensation A' to be invoked. Compensation B' will not be invoked at this stage as it is on compensation task 2 and only compensation task 1 is invoked by the first reversal operator. After the first reversal, activity C is performed. Reversal is then invoked on compensation task 2 which causes C' followed by B' to be executed.

The compensation information of a process is maintained by a compensation function that for each compensation task index, returns the associated compensation process. When the primary task of a compensation pair concludes its execution, the compensation task is composed in sequence with the original compensation process for that task.

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. When merging compensation tasks, those tasks are merged in parallel. In the process

$$(A \div_1 A'); (B \div_2 B'); \{1, 2\} \rhd 3$$

the merge operator will compose compensation task 1 (A') and compensation task 2 (B') in parallel and add the result ($A' \parallel B'$) to the front of compensation task 3. Tasks 1 and 2 will be cleared.

The StAC process

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

can be represented in StAC$_i$ as

$$(A \div_1 A'); (B \div_2 B'); \boxtimes_2; (C \div_2 C'); \{2\} \rhd 1$$

Here the inner compensation scope is represented as a 'new' compensation task. When the reversal instruction is invoked on compensation task 2 it will only execute B'. Compensation process A' that in StAC was outside the braces, 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.

Utilising the facility of multiple interleaved compensation tasks, [10] introduced the *selective* compensation and *alternative* compensation. With selective compensation, the reversal selects some activities to be compensated, while preserving the compensations for other activities. With alternative compensation, several alternative compensation tasks may be attached to an activity and the reversal selects one of these alternatives for invocation. These two multiple compensation mechanisms are discussed in detail in [10] where selective compensation is used in a travel agency business process to compensate those parts of a client itinerary that fail to get reserved.

5.2 Modelling Named Compensation in BPEL

The multiple compensation provided by StAC$_i$ allows us to model named compensation in BPEL. We write named compensation in BPEL as $compensate(S)$, where S is the name of the scope to be compensated. The meaning of the $compensate(S)$ activity is to run any compensation handler installed for scope S. To deal with this, we associate a compensation task with each scope, identified using the name of the scope. Now a compensation handler for scope S is installed on a compensation task labelled by S and that compensation can later be invoked by compensating task S. Consider the nested scopes illustrated in Figure 5 with an outer scope $S1$ and inner scopes $S2$ and $S3$. This will give rise to three compensation tasks, $S1$, $S2$ and $S3$. When $A2$, the body of scope $S2$, completes successfully, then compensation handler $CH2$ will be installed in compensation task $S2$. Similarly for $S3$ and $S1$. Fault and compensation handlers $FH1$ and $CH1$ may contain $compensate(S2)$ or $compensate(S3)$ activities and these will be translated into the corresponding compensate operations in StAC$_i$.

As well as adding named compensation to our BPEL semantics, we will also continue to model name-free compensation. For example, $FH1$ in Figure 5 may contain a name-free *compensate* activity. In this case both $S2$ and $S3$ should be compensated provided their compensation handlers have been installed. To

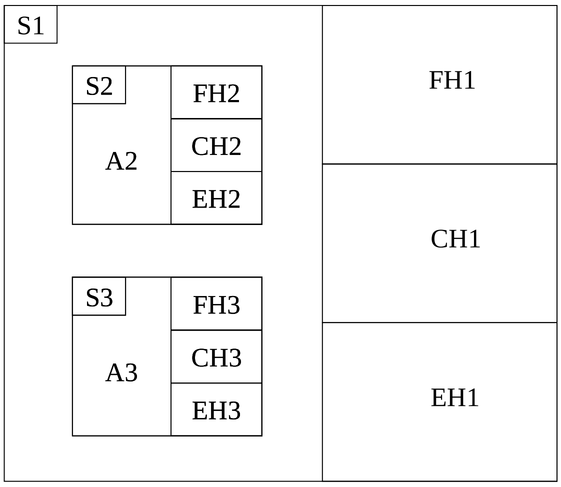

Figure 5: Example nested BPEL scopes.

model name-free compensation, as well as installing $CH2$ on compensation task $S2$, we will also introduce a compensation task $S1'$ to model default compensation for $S1$ ($S1'$ is a distinct identifier from $S1$ and all other scope identifiers and is used to identify the default compensation task for scope $S1$). In the example of Figure 5, both $CH2$ and $CH3$ would be installed on $S1'$. Now a name-free compensate activity in $FH1$ or $CH1$ is translated into compensation of task $S1'$.

The installation of $CH2$ and $CH3$ on compensation task $S1'$ also deals with default compensation. Recall that default compensation arises when a scope has no defined compensation handler or when the compensation handler is not installed because of an exception. With our proposed scheme, if $CH1$ is not present or does not get installed because of an exception, then the default compensation task will be used if $S1$ needs to be compensated.

To recap, given a scope $S2$ which is directly contained within scope $S1$, when the body of $S2$ completes successfully, then $CH2$, the compensation handler for $S2$, is installed on two compensation tasks, $S1'$ and $S2$. This double installation in modelled in StAC$_i$ as follows:

$$((A2 \parallel EH2) \div_{S2} CH2) \div_{S1'} CH2$$

This translation of a scope requires two scope names, that of the scope being translated ($S2$) and that of the immediately enclosing scope ($S1$). To deal with this, we add a scope name as an extra argument to the translation function: $\mathbb{T}(A, S)$ translates BPEL activity A contained within scope S. All of the translation rules for BPEL activities given in Section 4.3, except those for scopes and compensate activities, are modified so that they simply pass this argument on to recursive calls of $\mathbb{T}$, e.g.,

$$\mathbb{T}(\ \mathit{sequence}(A_1, \ldots, A_n),\ S\ )\ =\ \mathbb{T}(A_1, S)\ ;\ \cdots\ ;\ \mathbb{T}(A_n, S)$$

We replace the translation rules for BPEL scopes given in Section 4.3 by the following rule. On successful completion of the scope, the compensation handler is installed in two places as discussed above:

$$\begin{array}{l}
\mathbb{T}(\ \mathit{scope}(S2, A, EH, FH, CH),\ S1\ )\ = \\
\qquad \mathit{skip}\ \{\ ((A' \parallel EH') \div_{S2} CH') \div_{S1'} CH'\ \}\ FH' \\
\textbf{where} \\
\quad A'\ =\ \mathbb{T}(A, S2) \\
\quad EH\ =\ \mathit{eventHandler}(\ \mathit{guardedActivity}(R_1, A_1), \ldots, \\
\qquad\qquad\qquad\qquad \mathit{guardedActivity}(R_n, A_n)\ ) \\
\quad EH'\ =\ \mathit{LOOP}(\mathbb{T}(R_1, S2); \mathbb{T}(A_1, S2))\ \parallel \cdots \parallel \\
\qquad\qquad \mathit{LOOP}(\mathbb{T}(R_n, S2); \mathbb{T}(A_n, S2)) \\
\quad FH\ =\ \mathit{faultHandler}(\mathit{catch}(f_1, F_1), \ldots, \mathit{catch}(f_n, F_n)) \\
\quad FH' = \mathit{dCH}\ ;\ (\ f_1 \Longrightarrow \mathbb{T}(F_1, S2)\ []\ \ldots\ []\ f_n \Longrightarrow \mathbb{T}(F_n, S2)\ ) \\
\quad \mathit{dCH}'\ =\ \mathit{skip} \div_{S2} (\boxtimes_{S2'}) \\
\quad CH\ =\ \mathit{compensationHandler}(C) \\
\quad CH'\ =\ \mathbb{T}(C, S2)
\end{array}$$

Notice that the first step (dCH) of the fault handler FH' is to install an invocation of the default compensation handler on compensation task $S2$. This is because in the case of a fault, the given compensation handler CH will not be installed and the default compensation will apply instead.

If the fault handler is not present, the above definition of FH' should be replaced by the following:

$$FH'\ =\ []e \in \mathcal{E} \cdot\ e\ \Longrightarrow\ (\ \boxtimes_{S2'}\ ;\ \odot_e\ )$$

If the compensation handler is not present, the above definition of CH' should be replaced by the following:

$$CH'\ =\ \boxtimes_{S2'}$$

A named BPEL compensate activity is translated to a StAC compensate operator indexed by the named scope, while a name-free compensate is indexed by the default handler for the containing scope:

$$\begin{aligned}
\mathbb{T}(\mathit{compensate}(S2), S1)\ &=\ \boxtimes_{S2} \\
\mathbb{T}(\mathit{compensate}, S1)\ &=\ \boxtimes_{S1'}
\end{aligned}$$

The translation scheme just described allows us to model named and name-free compensation invocation. However, the semantics of default compensation in this new scheme is not quite the same as in the scheme defined in Section 4.3. With the previous scheme, default compensation was based on the StAC approach which is that compensations for sequentially composed scopes are executed in reverse order while compensations for scopes composed in parallel can be run in parallel. In the new scheme, the compensation handlers will be run in reverse order of their installation. This is exactly the definition for default compensation order in BPEL [12].

6 Conclusions

The combination of explicit and implicit compensation supported by BPEL is quite complicated and it is unclear to what extent this complexity is required for application to business transactions. A downside of the complexity is that it may hinder reasoning, both formal and informal, about BPEL designs. [8] describes a trace semantics for a simplified version of the StAC language. In that work, the simplified language does not contain $\boxtimes$ or ☑. Instead the invocation of installed compensations for a transaction block is automatic in the case of an exception in the block while installed compensations are discarded in the case of successful termination of a transaction block. While this is more restrictive than StAC and BPEL, it does lead to a cleaner compositional semantics and a language that appears to be more amenable to modular verification than StAC or BPEL.

Bruni et al [4] have developed an operational semantics for a language with similar operators to StAC, including compensation pairs and transaction blocks (or sagas as they call them). Like the work of [8], and unlike StAC, the invocation of compensation in a saga is automatic depending on failure or success which leads to a neater operational semantics.

Other researchers have worked on formalising notions of compensation. In [18], a compensation is formalised in terms of the properties it has to guarantee. However, [18] does not provide a modelling language as StAC does, rather it provides a characterisation of properties of compensation. ConTracts [23] 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.

Recently Misra [21] has developed the *Orc* language for so-called wide area computing. This language provides operators for service composition and the intended application is the orchestration of web services. The language is declarative in nature with a strong emphasis on algebraic properties of the operators. A key idea of Orc is the use of angelic nondeterminism to model multiple ways in which a service might be achieved. Such angelic nondeterminism could be implemented using speculative parallelism whereby multiple possible ways of achieving a service are attempted in parallel. The link between compensation and speculative parallelism has been investigated in [8] where it show that compensation can be used to 'tidy-up' those speculative branches that fail or are not availed of.

We believe that the combination of a process oriented notation such as StAC with a state oriented notation such as B, provide a very rich way of describing both the orchestration and the data aspects of long running business transactions. Compensation is an important feature of long running transactions and we believe that StAC was the first process oriented language to introduce compensation operators. Because of the closeness of StAC to BPEL we were able to define a straightforward mapping from BPEL to StAC. We believe this mapping makes the semantics of BPEL activities more accessible as well as giving them a formal semantics. This suggests that combinations such as StAC and B provide a good basis for specification of contracts and formal analysis of business transactions.

Acknowlegdements

We are grateful to the anonymous reviewers for very valuable feedback which has lead to improvements in the paper.

References

1. J.R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. A. Avizienis, J.-C. Laprie, B. Randell, and C.E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.
3. D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web services architecture. `www.w3.org/TR/ws-arch/`, 2004. W3C Working Group, Note 11.
4. R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL 2005*, 2005.
5. M. Butler. csp2B: A practical approach to combining CSP and B. *Formal Aspects of Computing*, 12:182–198, 2000.
6. 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.
7. M. Butler and C. Ferreira. An operational semantics for stac, a language for modelling long-running business transactions. In *Coordination 2004*, volume 2949 of *LNCS*. Springer-Verlag, 2004.
8. M. Butler, C. Ferreira, and C.A.R. Hoare. A trace semantics for long running transactions. In *25 Years of CSP*, 2004.
9. 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.

10. 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.
11. 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.
12. 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.
13. C. Ferreira. *Precise Modelling of Business Processes with Compensation.* PhD thesis, University of Southampton, 2002.
14. 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.
15. H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of ACM SIGMOD*, pages 249–259, 1987.
16. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, 1993.
17. C.A.R Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
18. H. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *16th VLDB Conference*, Brisbane, Australia, 1990.
19. 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.
20. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
21. J. Misra. Computation orchestration – a basis for wide-area computing. In *2004 NATO Summer School, Marktoberdorf*, 2004.
22. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, September 1981.
23. H. Wachter and A. Reuter. The ConTract model. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications.* Morgan Kaufmann Publishers, 1992.

A Semantics of Parameterised Throw

The transition rules dealing with early termination are easily extended to deal with parameterised throw. For most rules, one just has to add the exception identifier to $\odot$ and *early* operators. For example, consider the non-parameterised rule below:

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

This rule is extended by adding the exception identifier f to operators $\odot$ and *early* stating that exception f has occurred:

$$\frac{}{(\odot_{\mathtt{f}},\ C,\ \sigma)\ \xrightarrow{\odot_{\mathtt{f}}}\ (early_{\mathtt{f}},\ C,\ \sigma)}$$

The only rule that cannot be extended by parametrisation of the exception operators is the rule below:

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

In this case the rule has to be replaced by two new rules. The rule on the left-hand side states that, if an attempt block terminates prematurely by the occurrence of exception f and there is a handler (R_f) for that exception, then the exception handler R_f will be executed. While the rule on the right-hand side, models the situation where no handler exists for exception f causing that same exception to be thrown again.

$$\frac{f \in E}{(P\{early_f\}(\;[]e \in E \cdot\; e \Longrightarrow R_e\;),\, C,\, \sigma) \;\xrightarrow{\tau}\; (R_f,\, C,\, \sigma)}$$

$$\frac{f \notin E}{(P\{early_f\}(\;[]e \in E \cdot\; e \Longrightarrow R_e\;),\, C,\, \sigma) \;\xrightarrow{\tau}\; (\odot_f,\, C,\, \sigma)}$$