Extending the concept of transaction compensation

by M. Chessell
C. Griffin
D. Vines
M. Butler
C. Ferreira
P. Henderson

The ability to compensate for previous activities, often in the case of failure or exceptional events, is an important feature of long-running business transactions. In this paper, we present several extensions to existing notions of compensation for business transactions. The extensions are described using a business process modeling language called StAC (Structured Activity Compensation) but are also placed in the context of IBM's BPBeans (Business Process Beans) enterprise technology. The meaning of the compensation mechanisms is made precise, as are issues of compensation scoping in multilevel transactions. The compensation extensions result in flexible and powerful mechanisms for modeling and implementing long-running business transactions.

To compensate is "to make amends for, to make up for."[1] In the context of business transactions, a compensation is an action taken when something goes wrong or when there is a change of plan. For example, when an airline has overbooked a flight and too many passengers turn up at the gate, something has gone wrong. The airline needs to take corrective action to resolve the problem. In this case, the airline will typically attempt to encourage some passengers to delay their journey by offering monetary payments. The payments and the rebooking of the flight are a compensation for the inability to seat these passengers on this flight.

In this paper, we present some extensions to the standard notion of compensation. We show that these extensions provide powerful and flexible mechanisms for modeling and building extended business transactions. Some of the mechanisms described here have been implemented as part of IBM's *BPBeans*[2] (Business Process Beans) technology. BPBeans is now part of the IBM WebSphere* Application Server Enterprise Edition.

The standard approach to compensation involves associating a compensation activity with the primary activities of a transaction.[3] If compensation is required, the compensation activities of all successfully executed primary activities are executed. In the approach of Reference 3, the compensation activities are expected to undo the effect of the primary activity to which they are associated. 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*.

We present a simple business process modeling language called *StAC* (Structured Activity Compensation), which incorporates compensation constructs. We use StAC as a vehicle for describing our extensions to transaction compensation. We present extensions of two types:

- The first type involves mechanisms that are more general than the standard concept of compensa-

0018-8670/02/$5.00 © 2002 IBM

tion. Compensation takes place at the application level. The standard concept of compensation is extended to include transactions that are not atomic, and thus compensation is not necessarily a semantic "undo." Compensation of sequential and concurrent activities is supported and compensation is hierarchically structured.
- The second type involves new mechanisms supporting multiple compensation. We introduce the notions of *selective compensation* and *alternative compensation*. In selective compensation, the reversal selects which activities should be compensated. In alternative compensation, several compensations may be attached to a primary activity and the reversal chooses which of these should be invoked.

We proceed with an overview of the standard approach to compensation. We then present the two types of extension to the standard approach.

Transactions and failure

Deciding what to do when things go wrong is one of the most difficult aspects of software design and development. Many mechanisms have been developed to help with the design and coding of error detection and correction. The most basic mechanisms involve the use of return codes and exception handling. These permit the program to detect and indicate that an error condition has occurred, but they do nothing to help the program with corrective action.

One of the first mechanisms to be introduced in order to help with corrective action is that of ACID transactions. In general in this paper, by "transaction" we mean a long-running transaction that is typically not ACID. When we are referring to an ACID transaction, it will be clear from the context.

ACID transactions. The ACID transaction is a concept that was first introduced during the 1960s (although the term ACID was not introduced until 1983.)[3] An ACID transaction is a grouping of actions or operations that together have the following properties:

Atomicity. The group of actions occurs atomically, that is, the actions either all happen or none happen.
Consistency. The group of actions together are a correct transformation of the state of the system.
Isolation. Even though transactions are processed concurrently, it appears to each transaction that other transactions occurred either before or after it.
Durability. Once a transaction "commits," its effects survive any system failures.

ACID transactions aid programmers immensely by allowing them to rely on a transaction processing monitor (a program that manages and coordinates the transactions in a system) to provide facilities for making a group of actions atomic.

The ACID transaction processing monitor normally provides services to an application to allow it to demarcate the work it wishes to perform into ACID transactions. The application makes a *begin* call at the start of the ACID transaction and then either a *commit* call (when it wants the actions to occur) or a *rollback* call (when the application wishes to abort the actions) at the end of the ACID transaction. This ACID mechanism assumes transactions are fast and simple. This may not be the case for complex operations. So, for example, exclusive locks put on data may be held for a long time, seriously reducing the throughput of the system. Also, ACID transactions involving a large amount of work can be very expensive to roll back, especially in the latter stages of their life. For example, think of a payroll application. If it were operating as a single ACID transaction, a server failure that occurred during the payment of the last employee would undo the successful payment of all other employees.

Compensation. Compensation is the act of making amends when something has gone wrong or when plans are changed. Although compensation can be as simple as undoing an original action (for example crediting an account that has been debited), it should not be viewed as simply *undoing* the original action. For example, if a bill has been sent to a customer, it is not possible to undo the sending. In this case the compensation would probably involve either sending an apologetic letter asking the customer to disregard the bill, or providing a refund.

Generally, a long-running transaction can be broken into smaller transactions and compensating transactions, which can then be combined into a form of extended transaction known as a "saga."[4] The saga is composed of a sequence of transactions, each of which (except for the very last one) has an associated compensation. If one of the transactions in the sequence reverses, the compensations associated with those transactions that were successfully completed are run in reverse order.

By using sagas, a long-running transaction can be broken into smaller pieces, each of which can keep fewer exclusive locks, and release those locks much earlier than the original long-running transaction. However, the saga violates the isolation principle of ACID transactions. If another transaction examines data being changed by the saga, it may see data from the middle of the execution of the saga.

Introduction to business process beans

In its simplest terms, the BPBeans framework allows customers to build Java** objects that represent their business processes. A business process models a particular piece of work that is useful to the business. Generally, this work involves inputting data, processing the data, updating some stored data, and producing a result.

To be useful, business processes are often connected together so that the output from one process becomes the input for the next. The network of business processes for a business function is usually defined in a business process model. These models can often be complex, especially when error handling and exception processing is included. To make them comprehensible, most business process models are arranged in a hierarchy of abstractions. At the top level of the model one can see the major business functions. Each business function can be expanded to show its main internal business processes, which may in turn be expanded through many levels until the simple primitive operations are exposed.

Although business process modeling is very useful for business managers, it is not straightforward to take parts of the model and implement them across a number of computer systems. This is because a business process model normally uses many different styles of business process, each of which requires a different piece of middleware technology. For example, Component Broker[5] can run customer-written objects that communicate with one another in a synchronous manner. WebSphere MQ*,[6] on the other hand, is very good at asynchronous message passing. A business process model is likely to be implemented using both styles of communication, and an implementation of part of a business process often involves integrating different types of middleware that use different terms and modes of operation.

BPBeans provides the means, in the form of the *ABC*[2] (Application Builder for Components) tool, for an application designer to build an application based on a business process model that makes use of different styles of processing (e.g., parallel or sequential processes communicating either synchronously or asynchronously). The BPBeans run-time environment is then responsible for combining the necessary middleware to support the application. In addition, the run-time environment will control transactions and advanced error recovery mechanisms such as compensation through properties and constructs added to the business process model. This means the programming interfaces that the customer-written code uses must be very simple.

Not only is the programming of the system based on the contents of the business process model, the deployment, monitoring, and debugging is also driven by the business process model. This means the organization works with a single view of the system.

BPBeans applications. A BPBeans application is made up of a hierarchy of nested components. At the bottom of this hierarchy are the primitive components. A primitive component contains a simple Java bean. This Java bean is loaded into the ABC tool, which creates some XML (Extensible Markup Language) that describes the services required by the Java bean. It is this combination of the XML and the Java bean that makes the BPBean.

From the ABC tool, the application designer is able to pull these primitive components together into composite components called processes and connect them. Processes may also contain other processes, creating a hierarchy of components. The BPBeans run time provides implementation for a number of useful primitive components, plus some process patterns such as the following:

- Concurrent processes that can support an arbitrary number of communicating tasks (activities or processes) running in parallel
- Sequential processes that step through a sequence of tasks, one at a time, using the result (outcome) of the previous task to determine which task to run next
- Compensation pair processes that combine two tasks together, where one of the tasks is run if compensation for the effects of the other task is required

The BPBeans framework also provides for acceptance of tasks and for reversal of tasks. The BPBeans framework uses a graphical representation of these patterns as shown in Figure 1. In this figure, the ovals

Figure 1 BPBeans example

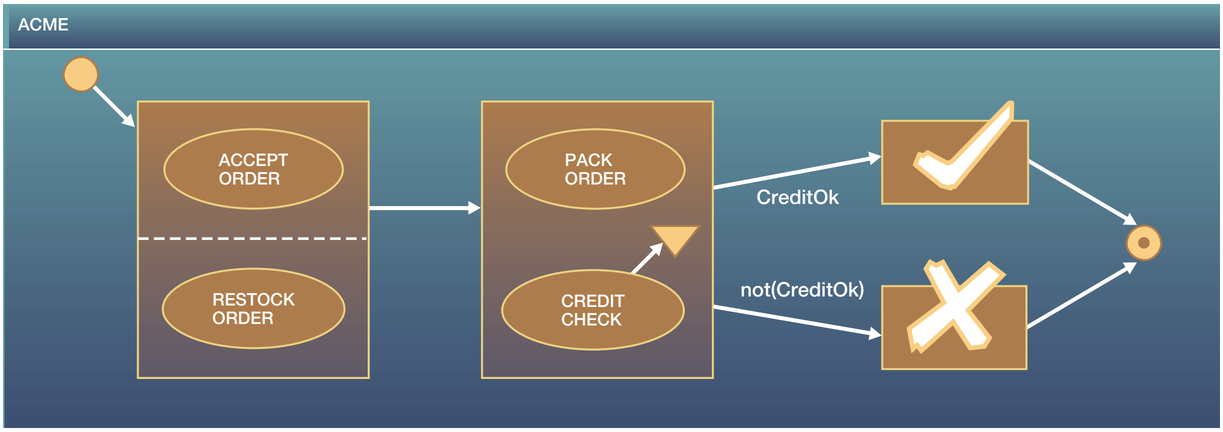

Table 1 StAC syntax

Syntax	Meaning
Process ; *Process*	Sequential processes
Process ‖ *Process*	Concurrent processes
PAR *i* IN *S* DO *Process*	Generalized concurrency
IF *Condition* THEN *Process* ELSE *Process*	Conditional
⊙	Early termination
{*Process*}	Termination scoping
Process ÷ *Process*	Compensation pair
[*Process*]	Compensation scoping
☑	Accept
☒	Reverse

represent activities, whereas boxes and arrows are used to group these. The box with the dashed line represents a compensation pair. In this case, *Accept Order* is compensated by *Restock Order*. Arrows represent sequencing of activities, so *Accept Order* takes place before *Pack Order* and *Credit Check*. Activities in the same box that are not connected by arrows represent concurrent activities, so *Pack Order* and *Credit Check* take place concurrently. The triangle to which *Credit Check* is attached means that the outcome used to select subsequent behavior is determined solely by *Credit Check* and is not affected by *Pack Order*. The box with the check represents transaction acceptance, whereas the box with the X represents transaction reversal. The solid circle represents the entry point, whereas the circle with the dot in the center represents the exit point. A process in BPBeans may consist of several hierarchically structured diagrams.

Structured activity compensation

StAC (Structured Activity Compensation) is a textual business process modeling language introduced in Reference 7. StAC supports sequential and concurrent activities, as well as compensation. We give an overview of the language in this section. Some of the language used for describing processes is explained in Table 1.

Sequential and concurrent activities. An activity in StAC corresponds to a primitive component in the BPBeans framework. Activities act on a global set of variables shared by all activities in a model. As in BPBeans, activities may be composed sequentially or concurrently using the sequential and concurrent operators.

The sequential construct is a binary operator that composes two processes, $P;Q$. In the process $P;Q$, P is executed first. When P completes, Q is executed.

There are two forms of concurrent construct, the binary form, $P \| Q$, which composes two processes in parallel, and the generalized form, PAR i IN S DO P, which models concurrent invocation of multiple instances of a process. For example,

$$\text{PAR } i \text{ IN } 1\,..\,10 \text{ DO } i.P$$

represents 10 concurrent instances of process P, where each instance of P is indexed by a unique number i in the range 1 . . 10. A concurrent process completes when all the constituent processes complete.

The sequential operator is associative, that is, $(P; Q);R = P;(Q;R)$, which means that we can write a nested sequential composition of the form $P;(Q;R)$ without parentheses as $P;Q;R$. Similar principles apply for the binary parallel operator.

Because activities act on a shared global variable set, the processes of a concurrent composition can interact indirectly via variables. A fuller description of StAC, including a description of the approach used to specify global variables and changes to global variables, as well as a formal semantics of StAC, may be found in Reference 7.

Early termination in StAC. The process terminator, $\odot$, causes a process to terminate early. The behavior that is made to terminate is limited by the termination scoping brackets, $\{\ldots\}$. For example, the process

$$\{P;\ \odot\,;\ Q\};\ R$$

will first execute P, then the terminator will cause the process within the braces to terminate so that Q will not get executed. The overall process will then continue by executing R. Termination scoping may be nested. In the case of concurrent processes, a terminator within one process also applies to the other process. For example, in the process

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

the terminator causes R to terminate. The terminator does not cause S to terminate because S is outside the termination scope. It is important to note that R may not terminate immediately on invocation of the terminator but at some later stage. This is because termination of concurrent processes would be implemented by sending messages to the processes instructing them to terminate, and these messages will not be transmitted or acted upon instantaneously.

The rules for the process terminator are:

- Invocation of a terminator within a sequential process causes that process to terminate immediately.
- Processes within the scope of a terminator that are running concurrently to the terminator may continue to execute for several steps after invocation of the terminator before terminating either prematurely or at completion.

Compensation in StAC. A compensation pair $(P \div Q)$ is a grouping of two tasks, where P is the primary task, and Q is the compensation task. When a compensation pair runs, it runs the primary task. Once the primary task has completed, the compensation task is remembered. At a later stage, the compensation task may be invoked. The instruction to perform compensation is indicated by the reversal operator, $\boxtimes$. For example, consider the following process, where A and B are activities:

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

This process will perform activity A and remember the compensation B. The transaction reversal instruction will then cause compensation activity B to be executed.

A sequence of compensation pairs is compensated in reverse order, so the process

$$(A1 \div B1);\ (A2 \div B2);\ (A3 \div B3);\ \boxtimes$$

executes $A1$, $A2$, and $A3$ sequentially and then, because of the transaction reversal instruction, executes $B3$, $B2$, $B1$ sequentially.

Concurrent compensation pairs are compensated concurrently, so the process

$$((A1 \div B1)\|(A2 \div B2)\|(A3 \div B3));\ \boxtimes$$

executes $A1$, $A2$, and $A3$ concurrently and then, because of the reversal instruction, executes $B1$, $B2$, $B3$ concurrently.

The acceptance operator, ☑, indicates that currently remembered compensations should be forgotten because they will no longer be required. For example, the process

$(A1 \div B1)$; ☑; $(A2 \div B2)$; $\boxtimes$

performs $A1$ followed by $A2$ and then performs the compensation $B2$. Compensation $B1$ is not performed, because it has been removed by the accept instruction.

Once the acceptance or reversal steps have been performed, a process can continue on to other steps.

These steps can result in new compensation pairs being invoked. Also, once a compensation has been performed, the compensation will be cleared. If there are no compensation pairs in between two successive reverses, the second reverse will have no effect.

The StAC language permits nested compensation pairs, modeling the fact that a compensation task can itself be compensated. The following example shows a process with two levels of compensation:

$$A1 \div (A2 \div A3); \boxtimes$$

First, activity $A1$ is executed and the compensation pair $A2 \div A3$ is remembered as the compensation for activity $A1$. Next, the reversal instruction will cause the compensation pair $A2 \div A3$ to be executed, by executing $A2$ and remembering the compensation $A3$. The remembered compensation can be invoked by a later reversal instruction. Thus, $A1$ is compensated by activity $A2$, and $A2$ is compensated by activity $A3$. The implications of this construction for implementations are yet to be explored.

Scoping of compensation. The compensation scoping brackets of StAC are used to delimit the scope of the accept and reversal operators. Within a scope, a reversal instruction will only execute those compensation activities that have been remembered since the start of the scope. For example, the process

$$(A1 \div B1); [(A2 \div B2); \boxtimes]$$

executes $A1$, $A2$, and $B2$ sequentially. Compensation $B1$ is outside the scope of the reversal instruction and does not get invoked.

Within a scope, an accept instruction will only remove those compensation activities that have been remembered since the start of the scope. For example, the process

$$(A1 \div B1); [(A2 \div B2); \text{☑}]; \boxtimes$$

executes $A1$, $A2$, and $B1$ sequentially. Compensation $B2$ does not get invoked as it is removed by the accept instruction. Compensation $B1$ does not get removed by the accept instruction because it is outside the scope of the accept.

When the end of a compensation scope is reached, nonaccepted compensations will be maintained because they may be invoked by the outer level scope. For example, the process:

$$[(A1 \div B1)]; \boxtimes$$

executes $A1$, remembering $B1$. When the end of scope is reached, $B1$ is maintained because it has neither been invoked nor accepted. The reversal instruction then causes $B1$ to be invoked.

StAC extends BPBeans by allowing nested compensation. As mentioned at the end of the previous subsection "Compensation in StAC," a compensation can have any other process as a compensation in StAC, whereas in BPBeans this is not permitted. Furthermore, there are some modeling differences between StAC and BPBeans. In BPBeans, each level in a process hierarchy overrides the lower-level compensation: a BPBeans process P is modeled in StAC as $[P; \text{☑}]$. Also, each concurrent process in BPBeans has its own compensation scope so that the concurrent composition of P and Q in BPBeans is modeled in StAC as $[P] \| [Q]$.

Order fulfillment example

To illustrate the use of compensation, we model a fictitious scenario based around an order fulfillment process in StAC. ACME Ltd distributes goods that have a relatively high value to its loyal customers. To accept and fulfill an order, the company performs the following steps:

- An order is accepted from a customer.
- Once the order is accepted, the warehouse is asked to prepare the order for shipment. As part of the preparation, a courier is booked to collect the order.
- Simultaneously with the warehouse preparing the order, ACME Ltd does a credit check on the customer to verify that the customer can pay for the goods. The credit check is performed in parallel because it normally succeeds and in this normal case we do not wish to delay the order unnecessarily.
- If the credit check fails, fulfillment of the order is stopped.

Application example using StAC compensation mechanism. In the following, underlined identifiers represent basic activities, that is, processes that cannot be further decomposed. Other identifiers represent processes that we define subsequently.

Table 2 ABC example without compensation mechanism

ACME	=	$\underline{\mathit{AcceptOrder}}$;
		FulfillOrder;
		IF not(*okFulfillOrder*) THEN *Compensate*
FulfillOrder	=	{*WarehousePackaging*
		∥ ($\underline{\mathit{CreditCheck}}$; IF not(*okCreditCheck*) THEN ⊙)}
WarehousePackaging	=	*PackOrder*
		∥ ($\underline{\mathit{BookCourier}}$; *CourierIsBooked* := *TRUE*)
PackOrder	=	(PAR *I* IN *OrderItems* DO
		$\underline{\mathit{i.PackItem}}$; *i.ItemIsPacked* := *TRUE*)
Compensate	=	IF *CourierIsBooked* THEN $\underline{\mathit{CancelCourier}}$;
		(PAR *i* IN *OrderItems* DO
		IF *i.ItemIsPacked* THEN $\underline{\mathit{i.UnpackItem}}$);
		$\underline{\mathit{RestockOrder}}$

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

$$
\begin{aligned}
&ACME = \\
&\quad (\underline{AcceptOrder} \div \underline{RestockOrder});\ \mathit{FulfillOrder}; \\
&\quad \text{IF } \mathit{okFulfillOrder} \text{ THEN } \text{☑} \text{ ELSE } \boxtimes
\end{aligned}
$$

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. Finally, if the order has been fulfilled correctly, the order is accepted, otherwise the order is compensated by invoking the reversal. (*okFulfillOrder* indicates successful outcome of the *FulfillOrder* activity.)

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:

$$
\begin{aligned}
\mathit{FulfillOrder}\ =\ & \\
&\{\ \mathit{WarehousePackaging} \\
&\parallel (\underline{\mathit{CreditCheck}};\ \text{IF not} \\
&\quad (\mathit{okCreditCheck})\ \text{THEN}\ \odot\ )\}
\end{aligned}
$$

Notice that the termination scope includes the *WarehousePackaging* process so that a failed credit check results in a termination instruction being sent to that process. This will cause *WarehousePackaging* to terminate eventually, possibly before all the items in the order have been packed.

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

$$
\begin{aligned}
\mathit{WarehousePackaging} = (&\underline{\mathit{BookCourier}} \\
&\div\ \underline{\mathit{CancelCourier}}) \\
&\parallel \mathit{PackOrder}
\end{aligned}
$$

The compensation pair books the courier, with the compensation action being to cancel the courier booking. $\underline{\textit{CancelCourier}}$ results in a second message being sent to the courier rather than reversing the sending of the original message. The *PackOrder* process packs each of the items in the order in parallel. Each $\underline{\textit{PackItem}}$ activity is compensated by a corresponding $\underline{\textit{UnpackItem}}$:

$$
\begin{aligned}
\mathit{PackOrder} = (&\text{PAR } i \text{ IN } \mathit{OrderItems} \\
&\text{DO } \underline{i.\mathit{PackItem}} \div \underline{i.\mathit{UnpackItem}})
\end{aligned}
$$

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

Application example without StAC compensation mechanism. A StAC model of the order fulfillment system that does not use the compensation mechanism is shown in Table 2. Here each primary activity sets a flag on completion indicating that it has been executed. The explicit *Compensate* process uses these flags to determine which compensation activities should be invoked. This style has a number of disadvantages. Extra variables need to be introduced to record which activities have taken place, and the application modeler needs to define the overall compensation behavior explicitly.

The most significant disadvantage of not using the compensation mechanism is that process reuse is severely hampered. In order to model the compensation mechanism explicitly, the application modeler needs to be aware of all activities that require compensation and what their compensation is. On the other hand, using the compensation mechanism provided by StAC, an application modeler can reuse an entire process definition, which may have compensation pairs embedded within it, without knowing what compensation is required. If reversal is required, the application modeler simply invokes the reversal operator, and the compensation mechanism of StAC ensures that the appropriate compensation is invoked on the reused process.

Multiple compensation

In this section we present some extensions to the StAC language that allow a process to have several simultaneous compensation tasks associated with it. A process decides to which task to attach compensation activities, and individual tasks can be accepted or reversed. This contrasts with the language presented in the previous subsection, "Scoping of compensation," where scoping is hierarchical and each scope has a single implicit compensation task.

To distinguish different compensation tasks, the compensation pair and the acceptance and reversal operators are indexed. The StAC language is extended as follows:

$Process \div_i Process$	Indexed Compensation Pair
☑$_i$	Indexed Accept
$\boxtimes_I$	Indexed Reverse

In the extended language, process $P \div_i Q$ has P as its primary task and, when P completes, compensation Q is remembered on compensation task i. The instruction to accept (i.e., clear) compensation task i is given by ☑$_i$, whereas the instruction to reverse (i.e., execute) compensation task i is given by $\boxtimes_i$. The compensation scoping brackets [ ] do not apply to the indexed compensation operators.

To help illustrate indexed compensation, consider the following example:

$$(A1 \div_1 B1);\ (A2 \div_2 B2);\ \boxtimes_1;\ (A3 \div_2 B3);\ \boxtimes_2$$

This process will invoke *A1* and *A2*. The reversal causes only compensation *B1* to be invoked. Compensation *B2* will not be invoked at this stage because it is attached to compensation task 2, and only compensation task 1 is invoked by the first reversal operator. After the first compensation, activity *A3* is performed. Reversal is then invoked on compensation task 2, which causes *B3* followed by *B2* 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. For example, consider the process

$$(A1 \div_1 B1);\ (A2 \div_1 B2)$$

After the execution of *A1*, *B1* is composed as the compensation process for task 1. When the primary task *A2* has completed, the compensation task *B2* is composed sequentially with compensation process *B1*. The resulting compensation for task 1 is the sequential process *B2*; *B1*. The reversal instruction for task i invokes the compensation process for task i, and the acceptance instruction for task i clears the compensation process for task i. The nonindexed version of StAC can be modeled by the indexed version. The scoping brackets (see the previous subsection, "Scoping of compensation") introduce a new compensation task with an empty compensation process, and all compensations within the brackets will be added to the compensation process of that new task. In the same way, all reverse and accept instructions within the brackets refer to the new compensation task. When the process within the brackets terminates, the compensation process of the new task will be composed in sequence with the compensation task of the surrounding process.

Utilizing the facility of multiple compensation tasks, we introduce two idioms of multiple compensation, *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 chooses one of these alternatives for invocation. We illustrate selective compensation through a travel agency example, and alternative compensation through a meeting scheduling example.

Selective compensation: Travel agency example. A travel agency (example taken from Reference 8) offers on-line trip reservation services to its clients. A client can compose an itinerary with several flight, car rental, and hotel reservations. The client is then asked to decide whether he or she wants to reserve an itinerary or to abort the reservation. Once the client's order has been confirmed, the reservations for the flights, car rentals, and hotels are made. Since these reservations are independent, they are made in parallel to speed up the overall process. If all the reservations in the client's itinerary are successful, the final itinerary is sent to the client, and this concludes the trip reservation process. Otherwise, if any of the reservations fail, the client is contacted and given the choice of selecting an alternative itinerary or aborting the reservation.

Before presenting the model of the travel agency, we introduce some extra constructs of the StAC language that are required for the example:

skip	Null activity
$\mathit{Process} \;\Box\; \mathit{Process}$	Choice
$\mathit{Process} * \mathit{Activity}$	Iteration

The process *skip* does nothing and completes immediately.

The choice between tasks P and Q is represented by $P \Box Q$. This represents a choice between the initial activities of P and the initial activities of Q and can be used, for example, to model a menu choice offered to a user. The initial activities of a process are the ones that can be executed immediately. For example, the initial activities of process

$$(A1; A2) \;\Box\; (B1; B2)$$

are activities $A1$ and $B1$, so the user has to choose between executing $A1$ or $B1$.

At the beginning of each iteration of the process $P{*}A$, the user has to choose either to execute activity A or process P. The selection of A terminates the iterations. If A is not selected, P is executed.

In the travel agency, a trip is arranged by getting an itinerary and continuing with the reservation:

$$\mathit{Trip} = \mathit{GetItinerary};\ \mathit{ContinueReservations}$$

Getting an itinerary involves continually iterating over offering the client the choice of selecting from a flight, a car, or a hotel until *EndSelection* is invoked:

$$\begin{aligned}\mathit{GetItinerary} = {} & (\underline{\mathit{SelectFlight}} \\ & \;\Box\; \underline{\mathit{SelectCar}} \;\Box\; \underline{\mathit{SelectHotel}}) \\ & * \;\underline{\mathit{EndSelection}}\end{aligned}$$

ContinueReservations starts by making the reservations on the client's itinerary. If some of the reservations failed, the client is contacted; otherwise, the process ends:

ContinueReservations =
MakeReservations;
IF *okMakeReservations* THEN *EndTrip*
ELSE ContactClient

The flight, car, and hotel reservations are made concurrently:

$$\begin{aligned}\mathit{MakeReservations} &= \mathit{FlightReservations} \\ &\ \| \ \mathit{CarReservations} \\ &\ \| \ \mathit{HotelReservations} \\ \mathit{FlightReservations} &= \mathrm{PAR}\ f\ \mathit{IN\ flights} \\ &\ \mathrm{DO}\ f.\mathit{FlightReservation} \\ \mathit{CarReservations} &= \mathrm{PAR}\ c\ \mathit{IN\ cars} \\ &\ \mathrm{DO}\ c.\mathit{CarReservation} \\ \mathit{HotelReservations} &= \mathrm{PAR}\ h\ \mathit{IN\ hotels} \\ &\ \mathrm{DO}\ h.\mathit{HotelReservation}\end{aligned}$$

The *FlightReservation* process reserves a single flight using the $\underline{\mathit{ReserveFlight}}$ activity. The travel agency uses two compensation tasks: compensation task S, representing compensation for reservations that have been booked successfully, and compensation task F, representing compensation for reservations that have failed. The choice of compensation task is determined by the outcome of the $\underline{\mathit{ReserveFlight}}$ activity.

Because we use two compensation tasks, instead of having a compensation pair we have a compensa-

Table 3 Travel Agency example without the StAC extensions

Trip	= . . .
GetItinerary	= . . .

ContinueReservations	= *MakeReservations*;
	IF *okMakeReservations*
	THEN ☑
	ELSE *ContactClient*

MakeReservations	= . . .
. . .	= . . .
f.FlightReservation	= *f.ReserveFlight* ÷ *f.DeleteFlight*

f.DeleteFlight	= IF *f.okReserveFlight*
	THEN $\underline{f.RemoveFlight}$
	∥ $\underline{f.CancelFlight}$
	ELSE $\underline{f.RemoveFlight}$

ContactClient	= $\underline{Continue}$; ☒; *Trip* □ $\underline{Quit}$; ☒

tion triple, with a primary task P and two compensations $Q1$ and $Q2$. We model this triple with a construction of the form:

$$P;\ \text{IF } c \text{ THEN } (\text{skip} \div_1 Q1) \text{ ELSE } (\text{skip} \div_2 Q2)$$

If P makes c true, this is equivalent to $P \div_1 Q1$ with $Q1$ being added to compensation task 1. If P makes c false, this is equivalent to $P \div_2 Q2$ with $Q2$ being added to compensation task 2.

The flight reservation and its associated compensations is defined as follows:

$$\begin{aligned} & f.FlightReservation = \\ & \qquad \underline{f.ReserveFlight}; \\ & \qquad \text{IF } f.okReserveFlight \\ & \qquad \text{THEN } skip \div_S (\,\underline{f.RemoveFlight} \\ & \qquad\qquad\qquad \parallel \underline{f.CancelFlight}) \\ & \qquad \text{ELSE } skip \div_F \ \underline{f.RemoveFlight} \end{aligned}$$

The $\underline{f.RemoveFlight}$ activity removes flight *f* from the client's itinerary. The $\underline{f.CancelFlight}$ activity cancels the reservation of flight *f* with the airline. The car and hotel reservations are defined similarly and are omitted here.

The *ContactClient* process is called if some reservations fail. The client is offered the choice between continuing or quitting:

$$\begin{aligned} ContactClient = \ & \underline{Continue};\ \boxtimes_F;\ Trip \\ & \square \\ & \underline{Quit};\ (\boxtimes_S \parallel \boxtimes_F) \end{aligned}$$

In the case that the client decides to continue, reversal is invoked on compensation task F, the failed reservations. This has the effect of removing all failed reservations from the client's itinerary. Compensation task S is preserved because the successful reservations may need to be compensated at a later stage. In the case that the client decides to quit, reversal is invoked on both compensation tasks. This has the effect of removing all reservations from the client's itinerary and canceling all successful reservations.

Finally, the trip reservation is ended by accepting both compensation tasks:

$$EndTrip = (\boxed{\checkmark}_S \parallel \boxed{\checkmark}_F)$$

In general, by selective compensation, we mean that some compensations can be reversed selectively, whereas the remaining compensations are maintained. We have modeled the selection criteria in the travel agency by using two compensation tasks and deciding immediately, when the primary process is complete, to which of these tasks to add the compensation. We then invoke the compensations selectively by selecting the appropriate compensation task.

An important feature of selective compensation is that those compensations that are not selected for reversal are preserved. This feature makes it difficult to model selective compensation in the subset of StAC that does not support interleaved compensation tasks (and difficult to implement in the current form of BPBeans).

Modeling the travel agency without the StAC extensions. Table 3 presents a StAC model of the travel agency without using multiple compensation tasks. Processes that are identical in both models of the travel agency are not described, for example, *Trip*, *GetItinerary*. The new model of the travel agency has a single implicit compensation task instead of having two compensation tasks, one for successful reservations and another for failed reservations. Here the compensation for *ReserveFlight* is the conditional process *DeleteFlight*. If the flight has been booked successfully, the compensation has to cancel the reservation and remove the flight from the client's itinerary. Otherwise, the compensation just removes the flight from the client's itinerary. If some reservations failed, the client is contacted, and in the case that the client decides to continue, reversal is invoked, causing the cancellation of all the services in the client's itinerary, including all successful bookings. This approach has the disadvantage of not retaining the reservations that were successful. Instead of allowing the client to replace just the part of the itinerary that failed with another alternative itinerary, the client has to choose the complete itinerary all over again.

Although it is possible to describe the travel agency without multiple compensation, the resulting model has a different behavior. It is not possible to model the cancellation of part of the itinerary, while maintaining the compensation information for successful bookings, with a single compensation task.

The behavior of the travel agency model without extensions is very similar to the travel agency example presented in Reference 8 (pages 259–274). In Reference 8 the authors use spheres of compensation to delimit the extent of the abort instruction. Aborting will only invoke compensations that are inside that sphere of compensation. Given that reservations are made concurrently, they have to belong to the same sphere of compensation, which causes the cancellation of the whole itinerary in the case of failed reservations.

With selective compensation it is possible to organize the compensation information into several compensations tasks, where each one of those tasks can later be reversed or accepted independently of the others.

Alternative compensation: Meeting scheduling example. In this example, the goal is to select a date for a meeting for which everyone in the team is available. A set of possible dates is proposed based on the availability of the meeting room. Every member of the team suggests possible dates from this initial set. If an agreement is reached, the meeting is scheduled; otherwise it will be canceled.

The top-level process is defined as a sequence of three processes. First, a set of possible dates on which the room is available is selected. Next, the team chooses possible dates for the meeting. Finally, a date is selected for the meeting and the meeting is scheduled.

$$ArrangeMeeting = CheckRoom; CheckTeam; \\ Decide$$

In this example, compensation is used in a novel way. Instead of the usual use of compensation when there is a failure or a change of plan, here compensation is used to perform a positive task. The arrange meeting application uses two compensation tasks: CF and CL. Compensation task CF represents activities that need to be confirmed, like the booking of the room or a date for the meeting. Compensation task CL represents activities that need to be canceled.

Process *CheckRoom* has a compensation pair within another compensation pair. In practice, it means that the date selection has two compensation activities: compensation $\underline{ConfirmRoom}$ in the task CF and compensation $\underline{CancelRoom}$ in task CL.

$$CheckRoom = (\underline{SelectPossibleDates} \\ \div_{CF} \underline{ConfirmRoom}) \\ \div_{CL} \underline{CancelRoom}$$

The $\underline{SelectPossibleDates}$ activity chooses a set of dates where the meeting room is available and temporarily books the room for those dates. The compensation activity $\underline{ConfirmRoom}$ will confirm the booking of a single date for the room and remove all the remaining dates. The compensation activity $\underline{CancelRoom}$ will remove all the dates temporarily booked.

Each member of the team suggests several dates for the meeting:

$$CheckTeam = \text{PAR } t \text{ IN } team \text{ DO} \\ (\underline{t.SuggestDates} \\ \div_{CF} \underline{t.ConfirmDate}) \\ \div_{CL} \underline{t.CancelDate}$$

In the $\underline{SuggestDates}$ activity, the member chooses his or her available dates from the possible dates for the meeting, and those dates will be inserted in the member's diary. The compensation activity $\underline{ConfirmDate}$ confirms the final date for the meeting and removes the remaining dates from the diary. The compensation $\underline{CancelDate}$ cancels all dates for the meeting in the diary.

The process *Decide* verifies that there is a date where all team members are available. In this case the booking of the meeting is confirmed, otherwise the meeting is canceled:

```
Decide =  IF $emptyDates$
          THEN $\boxtimes_{CL}$; $\boxed{\checkmark}_{CF}$
          ELSE $\underline{SelectDate}$; $\boxtimes_{CF}$;
          $\boxed{\checkmark}_{CL}$
```

When *emptyDates* is true, the meeting has to be canceled, as the set of dates acceptable to all team members is empty. This is achieved by reversing compensation task *CL* and accepting compensation task *CF*. The reversal of compensation task *CL* will remove the temporary bookings of the meeting room and clear the suggested dates from the team members' diaries. When *emptyDates* is false, compensation task *CF* is reversed, and *CL* is accepted. The reversal of *CF* will confirm the booking of the room and the meeting date on each member's diary.

The distinctive feature in alternative compensation is that activities can have several alternative compensation activities remembered for them simultaneously. Later a decision is made about which of the compensations attached to an activity should be invoked.

In this example, the compensation mechanism is used to perform a positive task and not just a compensating task. All the confirmations are performed by invoking the reversal on the compensation task *CF*. In this case, reversal is not invoked with the intention of correcting some failure.

Discussion

The following subsections review related work in the area of transaction compensation, discuss StAC extensions to compensation, relate compensation and ACID transactions, and compare compensation with exception-handling mechanisms.

Related work. In the *saga* construct introduced in Reference 4, transactions are nonhierarchical and purely sequential. Compensation activities are expected to undo the effect of the associated primary activity so that the atomicity of transactions is preserved, and compensation would normally be invoked when there is a failure in a system. In StAC, transactions are not atomic, since there is no requirement for compensation activities to semantically undo the effect of primary activities with which they are associated. Instead, the application or component developer must decide on the appropriate compensation to be associated with primary activities. The decision about whether and when to invoke compensation takes place at the application level rather than being based on system failure. This means that compensation can be used to achieve some desired behavior in the event of "nonfailing" outcomes, as well as to recover from failure.

In *nested transactions*[9] a transaction is decomposed into a hierarchy of subtransactions. Each subtransaction can either commit or roll back, and the "commit" will only take effect when its parent transaction (the transaction's predecessor in the hierarchy) commits. The rollback of a transaction causes all of its subtransactions to roll back. The tree structure of nested transactions creates a similar structure to the StAC compensation scoping: invoking an accept or reject instruction within a StAC compensation scoping will only affect the processes inside that scope; similarly invoking a commit or a rollback within a nested transaction will only affect its subtransactions.

A difference between StAC and nested transactions lies in the fact that in StAC the occurrence of an accept instruction in a compensation scope takes place immediately and is not dependent on the outcome of its predecessor in the hierarchy. Similarly to StAC, in *open nested transactions*,[10] which are a generalization of nested transactions, subtransactions can commit or abort independently of their predecessor. Considering that a transaction can be aborted after several of its subtransactions have already committed, open nested transactions require a compensation function for each subtransaction. The compensation function has to semantically undo the effects of committing its corresponding transaction. In both nested and open nested transactions the invocation of rollback is based on system failure, whereas in StAC, compensation is determined by the application. StAC provides a more precise definition of the nesting and scoping of compensation than nested or open nested transactions, as well as a more precise definition of the relationship between the execution of sequential and concurrent primary activities and their corresponding compensations. Besides that, StAC features, such as nonatomic compensation and multiple compensation tasks, are not represented in nested or open nested transactions.

A more formal approach that attempts to overcome the limitations of ACID transactions is presented in Korth et al.[11] The authors introduce the notion of compensating transactions; these transactions allow access to uncommitted data and undoing of committed transactions. Compensation is formalized in terms of the properties it has to guarantee: a compensating transaction has to reverse the effects of execution of the associated transaction, so that the state of the system after the compensation must be identical to the state before the execution of the transaction. This notion of compensation is very restrictive and for real-world actions (e.g., firing a missile, sending a letter) is impossible to achieve. Besides this, their approach does not provide a language as StAC does; instead the focus is on properties of compensation.

ConTracts[12,13] is a more structured approach to compensation. In ConTracts a system is described as a set of steps (actions or operations) that are executed according to a script (control flow description). Each step must have an associated compensation that will be invoked explicitly by the user within a conditional instruction: if the outcome of a step is false, then the associated compensation is executed. In this approach, a compensation step has to semantically reverse the effects of the associated step, which can be more than just undoing. Although compensations may not be atomic, each step can only have a single compensation. ConTracts does not have equivalent instructions to the StAC acceptance and reversal, and consequently in ConTracts compensation has to be explicitly invoked.

Reference 14 describes the basic constructs that a workflow specification language should support, namely sequence, iteration, splits (AND and OR), and joins (AND and OR). StAC supports those basic constructs directly. For example, workflow AND-split and OR-spit are represented in StAC, respectively, by parallel and choice constructs. Furthermore, StAC can also model most of the advanced workflow constructs described in Reference 15, such as implicit termination and multiple instances. This indicates that StAC is a suitable workflow modeling language with the advantage of having a formal semantics. Most workflow languages follow a transactional approach to recovery, which overlaps with the related work we already discussed. A different approach to recovery in the domain of workflow systems is presented in Reference 16. The authors' approach combines transaction atomicity with the concept of exception handling present in some programming languages such as C++ or Java. When an exception is raised, the signaler is replaced by an alternative activity, while the system has to undo all changes made by the signaler using spheres of atomicity. In this approach the overall process has to be atomic, so that it can be possible to semantically undo all its effects. A process specification has to verify several properties in order to guarantee its well-formedness. Because of the combination of exception handling and spheres of atomicity, these properties are complex and difficult to verify. As we show, exception handling can be formally modeled in StAC without the complexity of Reference 16.

StAC extensions to compensation. This section summarizes StAC (and BPBeans) extensions to compensation and at the same time highlights the distinctions between StAC and other languages that support compensation. We focus the comparison on the ConTract model, because it is the model with most similarities with StAC.

Nonatomic compensations. In both BPBeans and StAC, a compensation can be a complex process. StAC broadens the BPBeans functionality of compensation by allowing the use of nested compensation, so that compensation can itself be compensated. In ConTract compensation can be a complex process, but nested compensation is not permitted.

Compensation invocation at the application level. In BPBeans and StAC, the invocation of compensation is done at the application level instead of being based on the occurrence of a system failure. In ConTract compensation can be invoked at the application level, although it has to be made explicitly because ConTract does not have instructions equivalent to the StAC acceptance and reversal instructions.

Multiple compensation. The most distinctive feature in StAC is multiple compensation, which allows a process to have several independent compensation tasks. Neither ConTracts nor any other approach mentioned in the related work covers multiple compensation.

Compensation and ACID transactions. We consider the relationship between compensation and ACID transactions. In many cases, the basic activities of a long-running BPBeans transaction will themselves be ACID transactions. The isolation property of ACID transactions will be particularly important in the case of concurrent activities. For example, a basic activity may involve updating a database—which means that the basic activity should be isolated from other concurrent activities that access the same data until the basic activity has completed.

When specifying ACID transactions involving some complex business logic, it may be convenient to use the compensation mechanism as part of the ACID transaction. This is especially the case when compensation extends beyond an ACID transaction. For example, an ACID transaction that updates a database could also include the automatic sending of an e-mail during the transaction. The e-mail could be compensated by the sending of another e-mail. The compensation could extend beyond the ACID transaction in that the compensation e-mail might be sent after the ACID transaction has committed. Although embedded in an ACID transaction, the sending of the original e-mail is not itself transactional in nature, so allowing its compensation to extend beyond the ACID transaction is reasonable.

Compensation and exception handling. Since compensation can be used to deal with exceptions, it is instructive to compare the compensation mechanism with exception-handling mechanisms found in programming languages such as Java. In general, exception-handling mechanisms have three important features: a means of jumping out of the flow of control (the *throw* statement in Java), a means to define the scope of the jump (the *try* statement in Java), and a means to provide code to handle the occurrence of exceptions (the *catch* statement in Java). All of these features are present in StAC, with process termination providing a means of jumping out of the flow of control, and compensation providing a means of defining behavior that handles exceptions. There are, however, a number of differences between StAC and exception handling in programming languages.

In StAC, the termination mechanism is completely separate from the compensation mechanism in that the flow of control is exited using the termination statement, whereas the compensation behavior is invoked using the reversal operator. In programming languages, these functions are combined in the raising of an exception, which results in the flow of control being exited and the exception handling code being executed.

In StAC, the primary behavior and the compensation behavior are packaged together as compensation pairs, and the compensation mechanism invokes all the compensation activities as required. This is more difficult to achieve in programming languages when several compensation activities are required. For example, consider the StAC process

$(A1 \div B1); (A2 \div B2); \boxtimes$

Representing this behavior using exception handling would require code of the form:

```
try {A1;
     try {A2;
          throw e}
     catch(e) {B2; throw e}}
catch(e) {B1}
```

Here the compensation activity *B1* has been separated from the primary activity *A1*. Also, the sequencing of the exception handling needs to be made explicit by raising a further exception after *B2*.

Another important difference is that the termination and compensation mechanisms in BPBeans and StAC work across concurrent activities as well as sequential activities, whereas exception handling in programming languages only works within single sequential threads.

Conclusion

Compensation is an essential feature of many business processes and the compensation mechanisms provided by BPBeans and StAC allow compensation to be represented and considered as part of a high-level business process model. The mechanisms are powerful in that they automatically take care of remembering and sequencing compensation activities. As explained in the previous subsection, "Application example without StAC compensation mechanism," the compensation pair and the reversal mechanisms contribute to the reusability of business components by freeing an application developer from having to be aware of the compensations required by a reusable process component. This allows for flexibility in constructing models and systems.

BPBeans is a feature of IBM's WebSphere that supports the construction of business systems from Enterprise JavaBeans** and comes with a run-time environment that implements the compensation mechanisms. StAC is a modeling language that was developed in order to explore the semantics of compensation in a more rigorous way. The formal semantics of StAC is described in Reference 7. The design of StAC was originally based on the compensation mechanisms provided by a prototype version of BPBeans. The formal nature of StAC allowed some ambiguities to be identified and then clarified, especially scoping issues. These then led to improvements in the design of BPBeans.

The simplicity of StAC also allowed us to explore some more general forms of compensation, leading to the selective and alternative compensation idioms.

The case studies to which we have applied them suggest that they are useful concepts, and the addition of selective and alternative compensation to BPBeans is being considered.

Currently we are continuing to explore the relationship between ACID transactions and the use of compensation mechanisms within ACID transactions. Whether compensation can be used to implement all ACID transactions is an open question. We are also investigating the use of compensation for exception handling in programming. Compensation has the potential to provide a more modular approach to exception handling, as well as providing exception handling across concurrent activities.

Acknowledgments

We wish to acknowledge the support of the IBM Faculty Partnership program in funding the participation of the authors who are members of the University of Southampton Declarative Systems and Software group.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc.

Cited references

1. *Chambers Twentieth Century Dictionary*, W. and R. Chambers Ltd., Edinburgh (1972).
2. *WebSphere Application Server Enterprise Edition 4.0—A Programmer's Guide*, IBM Corporation (2002). See http://www.redbooks.ibm.com/redbooks/SG246504.html.
3. J. Gray and A. Reuter, *Transaction Processing Concepts and Techniques*, Morgan Kaufmann Publishers, San Francisco, CA (1993).
4. H. Garcia-Molina and K. Salem, "Sagas," *Proceedings of ACM SIGMOD*, San Francisco, CA (1987), pp. 249–259.
5. *Redbook—IBM Component Broker Connector Overview*, IBM Corporation (1998). See http://www.redbooks.ibm.com/redbooks/SG242022.html.
6. B. Blakeley, H. Harris, and R. Lewis, *Messaging and Queuing Using the MQI: Concepts and Analysis*, McGraw-Hill, Inc., New York (1995).
7. M. Butler and C. Ferreira, "A Process Compensation Language," *Proceedings of the Integrated Formal Methods 2000 Conference*, *Lecture Notes in Computer Science*, Vol. 1945, Springer-Verlag (2000).
8. F. Leymann and D. Roller, *Production Workflow Concepts and Techniques*, Prentice Hall, Englewood Cliffs, NJ (2000).
9. J. Moss, "Nested Transactions and Reliable Distributed Computing," *Proceedings of the IEEE Symposium on Reliability in Distributed Software and Database Systems*, Pittsburgh, PA. IEEE CS Press (1982).
10. G. Weikum and H. Schek, "Concepts and Applications of Multilevel Transactions and Open Nested Transactions," *Database Transaction Models for Advanced Applications*, A. Elmagarmid, Editor, Morgan Kaufmann Publishers, San Francisco, CA (1992).
11. H. Korth, E. Levy, and A. Silberschatz, "A Formal Approach to Recovery by Compensating Transactions," *Proceedings of the 16th Very Large Data Bases Conference*, Brisbane, Australia (1990).
12. A. Reuter, K. Schneider, and F. Schwenkreis, "ConTracts Revisited," *Advanced Transaction Models and Architectures*, S. Jajodia and L. Kerschberg, Editors, Kluwer Academic Publishers, Norwall, MA (1997).
13. H. Wächter and A. Reuter, "The ConTract Model," *Database Transaction Models for Advanced Applications*, A. Elmagarmid, Editor, Morgan Kaufmann Publishers, San Francisco, CA (1992).
14. *Workflow Management Coalition Terminology and Glossary* (WFMC-TC), Workflow Management Coalition, Hampshire, United Kingdom (1999).
15. W. van der Aalst, A. Barros, A. Hofstede, and B. Kiepuszewski, "Advanced Workflow Patterns," *Business Process Management: Models, Techniques, and Empirical Studies*, W. van der Aalst, J. Desel, and A. Oberweis, Editors, *Lecture Notes in Computer Science*, Vol. 1806, Springer-Verlag (2000).
16. C. Hagen and G. Alonso, "Exception Handling in Workflow Management Systems," *IEEE Transactions on Software Engineering* **26**, No. 10, 943–956 (2000).

Accepted for publication May 16, 2002.

Mandy Chessell *IBM United Kingdom Laboratories Ltd., Hursley Park, Winchester, Hants SO21 2JN (electronic mail: mandy_chessell@uk.ibm.com).* Mrs. Chessell is an IBM Master Inventor and member of the IBM Academy of Technology. She joined the Hursley Development Laboratory in 1987 and is now a software architect leading a team of developers producing business process modeling software for WebSphere. She is the first woman ever to win the Silver Medal of Engineering from the Royal Academy of Engineering. Two years ago, MIT's *Technology Review Magazine* voted her as one of the TR100, a group of people under 35 years of age most likely to make significant technical innovations in the twenty-first century. She has a M.Sc. degree in software engineering from the University of Brighton, England, received in 1997.

Catherine Griffin *IBM United Kingdom Laboratories Ltd., Hursley Park, Winchester, Hants SO21 2JN (electronic mail: cgriffin@uk.ibm.com).* Ms. Griffin is a software engineer at the IBM Hursley Development Laboratory in the United Kingdom working in the Transaction Systems organization. She joined the laboratory in 1993 and has worked on several of IBM's program products including CICS® for OS/2® and the CICS Transaction Gateway. She has a B.Sc. degree from the University of Nottingham, England, received in 1993.

David Vines *IBM United Kingdom Laboratories Ltd., Hursley Park, Winchester, Hants SO21 2JN (electronic mail: dvines@uk.ibm.com).* Mr. Vines is a software engineer at the IBM Hursley Development Laboratory in the United Kingdom working in the Transaction Systems organization. He joined the laboratory in 1984 and has worked on a wide variety of IBM's program products including GDDM®, MQSeries®, LANDP®, and, most recently, Component Broker and WebSphere. Mr. Vines is a joint inventor on eleven patent applications. He has a B.Sc. degree from the University of Exeter, England, received in 1984.

Michael Butler *Department of Electronics and Computer Science, University of Southampton, Highfield, Southampton, SO17 1BJ, United Kingdom (electronic mail: mjb@ecs.soton.ac.uk).* Dr. Butler is a professor of computer science at the Declarative Systems and Software Engineering group of the Department of Electronics and Computer Science at the University of Southampton. He received a B.A. degree in computer science from Trinity College, Dublin, in 1988, an M.Sc. degree in computing from the University of Oxford in 1989, and a D.Phil. degree in computing from the University of Oxford in 1992. His research is mostly centered around formal methods for software engineering. He works on verification and refinement techniques for distributed systems and control systems, on integration of formal methods with software engineering techniques such as UML, and on the development of tool support for refinement and verification. He has published many papers in this area and is on the program committees of several conferences on formal methods.

Carla Ferreira *Department of Electronics and Computer Science, University of Southampton, Highfield, Southampton, SO17 1BJ, United Kingdom (electronic mail: cf@ecs.soton.ac.uk).* Ms. Ferreira is a research assistant in the Declarative Systems and Software Engineering group. She received an M.Sc. degree in computer science in 1997 from the University of Minho, Portugal. Currently she is working on her Ph.D. thesis on precise modeling of business processes.

Peter Henderson *Department of Electronics and Computer Science, University of Southampton, Highfield, Southampton, SO17 1BJ, United Kingdom (electronic mail: ph@ecs.soton.ac.uk).* Dr. Henderson is Professor of Computer Science and head of the Declarative Systems and Software Engineering group at Southampton. His research interests include software engineering, business process modeling, and software architectures. He is a member of IFIP WG2.3 (Programming Methodology) and IFIP WG2.8 (Functional Programming). He leads several research projects on dependable software systems. Between 1996 and 2000 he was program coordinator for the UK EPSRC's managed research program, Systems Engineering for Business Process Change.