# CICS Business Transaction Services - enabling robust extended transactions

*Ian J Mitchell, IBM UK Laboratories, Hursley Park*
*ianj_mitchell@uk.ibm.com*

CICS Transaction Server 1.3 includes a significant set of new APIs under the umbrella title of *CICS Business Transaction Services* (CICS BTS). The aim of this set of APIs is to ease the development and deployment of complex, multi-transaction applications in the mainframe CICS environment. The APIs enable the explicit management of the collection of individual CICS transactions that implement an application.[1]
In this paper I will describe the programming model supported by CICS BTS, discuss some of the fundamental concepts that proved challenging, and finally compare CICS BTS with other models for extended transaction management.

## BTS Programming Model

Fundamentally, BTS introduces an event-driven programming model to CICS. This sounds deceptively simple - event-driven programming is not new, and has proved to be an extremely popular style in many application areas (GUI frameworks, for example). Indeed, it is possible to identify programming constructs in the existing CICS API that are event-driven - the so-called 'pseudo-conversational' style of application is essentially driven by interaction events. So what's different about BTS and its style of event-driven programming?

### What is BTS aiming to do?

The aim of BTS is to make it easier to develop complex, multi-transaction applications. So a BTS application is probably made up of more than a single CICS transaction. Finding applications like this is certainly not hard! Only the most simple business requirements can be supported by an application that entails a single set of resource updates in a single, short-lived Unit of Work.
So most CICS applications assemble individual CICS transactions into something useful to the customer's business. The business will recognise the composite function of the assembly as a *Business Transaction*, for example, a holiday booking. But, being a Transaction Monitor, CICS has previously left the mechanics of the assembly largely to the ingenuity of the application programmer.
Obviously the existing features in the CICS API have proved very useful in the assembly of multi-transaction applications. Some of the more useful features are:

---

[1]See *CICS Business Transaction Services - SC34-5268-00*.

- Pseudo-conversations - these are sequences of transactions that result from the interactions during a conversation with a user terminal device.
- The START command - enables one transaction to initiate another.
- Transient Data queue triggers - the presence of data may trigger a transaction.

Nevertheless, the job of controlling the collection of transactions that make up a Business Transaction has been a challenge for application programmers over the years. If the low-level control has been a difficult problem, then the task of specifying the pattern of transactions has been even harder. Application packages have been developed to try and ease the burden on the application programmer, but these can quickly become industry specific and they still have only the basic CICS functions available under the covers.
So how does BTS aim to help? BTS enables programmatic, flexible, transactional control of transactions.
**Programmatic** - it is possible to write application programs that specifically control the execution of transactions. The control operations are accessed through extensions to the translated CICS API thereby enabling CICS programmers to code BTS applications in the traditional CICS languages familiar to them.
**Flexible** - the pieces of the Business Transaction can execute in sequence or in parallel. The control structure can range from a simple sequence to a complex tree of interrelated entities.
**Transactional** - the control state and operations are managed in a fully recoverable manner.

## Application control of transactions

BTS introduces the ability to write programs that control transactions. Thus a program is able to determine how a set of transactions executes. The program can construct a simple sequence of transactions, or allow a set of transactions to execute in parallel.
It is important to point out that it is *functional decomposition* that is occurring here. The program is using the transactions that it controls to implement a composite function.
What does *control of a transaction* amount to? BTS allows the controlling program

- to create an entity representing the transaction
- to specify its attributes
- to initiate the transaction
- and to be informed of the transaction's outcome.

Of course, in CICS, all programs execute within a transactional environment. So the controlling application program is itself executing within a transaction and the control operations on other transactions are transactional. (The existing 'protected' form of START operates transactionally.) So it is better to describe BTS as enabling transactions to control transactions.

## What are BTS processes, activities and events?

The entity that the controlling transaction creates and through which control is exercised is called an *activity*. Data can be associated with activities by placing it in *containers*. Execution of the controlled transaction is initiated by *activating* the activity. It is natural to talk about *parent* and *child* roles for the activities.
Progress is signalled by driving activities with *events*. Events embody the reason for an activation. *Fired* events cause activations. When an activity is activated, the application program is obliged to ask the reason for the activation. Events are delivered to the child from the parent (termed *input events*), and events are delivered to the parent when the child completes (termed *completion events*). Containers can be used to pass data back and forth between parents and children.

Completion events are one of the distinguishing features of the BTS programming model. The existing START command provides a 'fire-and-forget' capability whereas BTS provides fully synchronised transaction control. I expand on the importance of completion events later.
The application program can specify that the activation of the child is either *synchronous* or *asynchronous* with the execution of the parent transaction. In the asynchronous case, the parent transaction must terminate and commit the control operation before the child activation will take place. Then the input event associated with the activation is delivered to the application program in a new transaction. When the child transaction completes, the child's completion event is fired in the parent. This case involves three transactions:

1. The transaction that is the activation of the parent that creates and requests the activation of a child.
2. The activation of the child in which an input event is fired.
3. The (re-)activation of the parent that is caused by the firing of the child's completion event.

So both parent and child activities must respond to the firing of events. This leads to the BTS activity application programs having the style of classic event-driven programs. There is a switch statement with cases to catch each event. This comprises the body of the event processing loop.
Whilst waiting for the next event to be fired, the activity is *dormant* and CICS stores the state of the activity on DASD in a so-called *Repository File*. Thus, processing resources are only consumed when an activity is actually processing an event.
There is no formal assignment of the role of parent or child in the BTS API. So a child can create and use its own children, and has to respond to their completion events. In this way the whole application can be decomposed into a tree of activities acting in parent and child roles. In general, an activity will receive a number of fired events and thus be activated in a sequence of activations as each event fires. An activity is deemed to be complete when all its children are complete, and no more events are pending.
The collection of activities in the tree is called a *process* in BTS terms. Of course, any tree structure will have a root. In the BTS case there is a topmost activity with no parent which is the *root activity*.

## Events, transactions and admitting failure

Both Transactional programming and Event-driven programming are mature styles of programming. BTS brings the two together.
ACID transactional management of application resources, such as file records and queue entries, is what CICS is all about. Message-driven processing is also well understood by CICS through inter-operation with products such as MQSeries. BTS events are another type of resource that is managed in a recoverable manner, and this places some obligations on the applications that make use of BTS's ability to control transactions.

### Event-handling - a mutual obligation

In asserting that BTS events are transactional, CICS has to manage them with care. Along with the rest of the state of a process or activity, the set of events it will respond to (and their status) is recorded in a logically recoverable VSAM file. Operations that change the state of events are thus dependent on the commital of the transaction that makes them.
The result of firing an event is a new transaction. The new transaction invokes the application program, which is obliged to process the fired event. Failure to process the event is an application error. An activity that suffers such an error is terminated, and its own completion event raised.

Events created by an application are safe in the hands of CICS - the application is obliged to treat them with a similar respect.

### Admitting failure

Events are the means by which progress is signalled to the application. In this context, failure is as significant as success. But failure of a transactional operation is usually accompanied by the backout of the recoverable updates - the aim of which is to hide the existence of the failed transaction. One of the essential achievements of BTS is to manage its events recoverably and yet to allow an event to signal failure as assuredly as it signals successful completion. (The scheme is currently the subject of a US Patent application.)
The scheme employed to achieve this aims to be as non-disruptive to the management of other recoverable resources as possible. The failed transaction is backed out in its entirety, but the backout processing for the event state causes *another transaction* to be initiated - this is called the *failure admission transaction*. This transaction's role is to record the failure by firing the completion event of the activity that has suffered the failure. The failing transaction and the failure admission transaction are interlocked to ensure that the failure information cannot be lost. The failing transaction does not complete its backout until the failure is safely communicated to the admission transaction. This involves handshaking between the tasks executing the transactions and the logging of the failure information by the admission transaction. Once the failure information is safely logged in the new transaction, the obligation to signal the failure is passed from the failed transaction and it can complete its backout. The admission transaction executes in the knowledge of the failure, but all other resource updates have been backed out. The admission transaction fires the completion event and commits forwards - so activating the parent activity in the normal way. The failure information is then available to the application program and it can behave accordingly.

## Using Processes, Activities, Events and Containers in extended transactions

Crucial to the design of a BTS application is deciding on the scope of the Process. The scope can be analysed in terms of the function of a Process and the control data it requires.

### The function of a process

BTS should reduce the gap between the form in which Business Processes are specified and the entities that appear in the application code. The mapping of the Business Process into business logic can be expressed directly in terms of dataflows in Containers and Events.

### The control data

The data managed as the state of a Process (in the Containers and the Events) is intended to be merely the data required to control the execution of the Process. An aim of BTS is to separate this control data from the true Enterprise Data (such as customer records, orders, accounts) that is best placed in a 'proper' database (for example, DB2). Processes and activities are not intended to be long-lived representations of enterprise data, rather they are the means to implement control of the transactions that update the enterprise data. The decomposition of a business into processes, and processes into activities is driven by functional analysis rather than data or object analysis. BTS containers are specialised resources intended for a localised purpose - the preservation of control data for extended transactions. They are not a suitable choice for data of wider interest to the business.

### Encapsulation and reuse

The formalisation of the activity as an entity provides increased opportunities for reuse of application code. An activity can be used to encapsulate an application function. The interface to an activity is essentially the set of events used to interact with it and the set of containers that pass data in and out. This allows activities to be reused in a 'plug-and-play' fashion across different applications.

### Scalability of BTS applications

The asynchronous form of activation provides opportunities for Workload Balancing decisions to optimise the CICS resources consumed by BTS applications. When activities are dormant their state is held offline in repository files. When an event causes an asynchronous activation, this can be dispatched on the most appropriate CICS region that has access to the repository file. This currently relies on VSAM Datasharing and is limited to a single OS/390 Sysplex.

## CICS BTS and the ConTract Model

The objectives of BTS are very similar to those expressed in the ConTract Model (Wachter & Reuter). It is interesting to examine some differences in the two approaches.

### Application coding style

BTS embeds the control logic operations into the traditional CICS API. This decision was the subject of a lot of debate and arguement. It avoids the invention of yet another programming/scripting language. It has been observed that even a COBOL program that confines itself to the business logic at the level typically present in scripting or rule-based languages can be as concise and readable as a higher level specification.
BTS activity programs can be as simple as that, or they can utilise the full power of a general purpose programming language. I consider the following quote as particularly persuasive - "With rules, you can't even ask the time of day!" (IJM 1997).

### Managing failure

The ConTract Model rightly identifies the implementation challenge of recoverably admitting failure. The admission must be made to the 'script manager'. The suggested mechanism is to employ nested transactions. The script executes in an outer transaction scope and always invokes the application in a nested transaction so that an application failure can be caught by the script's transaction.
CICS BTS does not require (nor implement) nested transactions. The application code is invoked in conventional, flat, CICS transactions. In the successful case, this is all that is required. As described above, in the (hopefully rare) case that failure must be admitted, the mechanism BTS employs is to back out the failed transaction and reliably initiate a transaction with the sole purpose of signalling the failure. I believe that this is a significantly more efficient scheme than relying on the overhead of nested transactions for all application execution.

### Where's Compensation, then?

A formalisation of Compensation is notably absent from the BTS programming model. This comes as somewhat of a surprise, especially when it is revealed that BTS itself was called 'Compensation Manager' at an early stage of development.

We considered Compensation at great length. But the more it was considered, the more it absented itself from the model. The essential function in lieu of Compensation is the recoverable admission of failure of activities.
If the application is structured into meaningful pieces, and business logic can rely on being informed of the failure of the pieces, then a useful compensation behaviour can be regarded simply as 'progress in an alternative direction'. In general, it was not observed that useful behaviour in a 'compensation phase' would be any less complex than that in the 'forward phase'. The reliable admission of failure enables the application to detect that compensation may be necessary. An activity can easily be structured to behave in alternative ways under the direction of the parent.

## Summary

BTS is a significant extension to the application management facilities provided by CICS Transaction Server. It addresses some well recognised problems facing the designer and deployer of enterprise applications. By introducing a single new, event-driven primitive - the activity - it provides a rich enough framework for multiple transaction management in which these problems can be overcome. The solution is embodied in an evolution of the partnership between the application program and the transaction monitor. In consequence, compensation management is seen as a natural extension of the application design, and not as a special mode supported by the system.