The CORBA Activity Service Framework for Supporting Extended Transactions

I. Houston[1], M. C. Little[2,3], I. Robinson[1], S. K. Shrivastava[3] and S. M. Wheater[2,3]

[1]IBM Hursley Laboratories, Hursley, UK
[2]HP-Arjuna Laboratories, Newcastle-Upon-Tyne,UK
[3]Department of Computing Science, Newcastle University, Newcastle-Upon-Tyne, UK

Abstract

Although it has long been realised that ACID transactions by themselves are not adequate for structuring long-lived applications and much research work has been done on developing specific extended transaction models, no middleware support for building extended transactions is currently available and the situation remains that a programmer often has to develop application specific mechanisms. The CORBA Activity Service Framework described in this paper is a way out of this situation. The design of the service is based on the insight that the various extended transaction models can be supported by providing a general purpose event signalling mechanism that can be programmed to enable activities - application specific units of computations – to coordinate each other in a manner prescribed by the model under consideration. The different extended transaction models can be mapped onto specific implementations of this framework permitting such transactions to span a network of systems connected indirectly by some distribution infrastructure. The framework described in this paper is an overview the OMG's *Additional Structuring Mechanisms for the OTS* standard now reaching completion. Through a number of examples the paper shows that the Framework has the flexibility to support a wide variety of extended transaction models. Although the framework is presented here in CORBA specific terms, the main ideas are sufficiently general, so that it should be possible to use them in conjunction with other middleware.

Key Words: CORBA, transaction, extended transactions, Web services, workflow

1. Introduction

Distributed objects plus ACID transactions provide a foundation for building high integrity business applications. The ACID properties of transactions (ACID: Atomicity, Consistency, Isolation, Durability) ensure that even in complex business applications the consistency of the application's state is preserved, despite concurrent accesses and failures. In addition, object-oriented design allows the design and implementation of applications that would otherwise be impractical. However, it has long been realised that ACID transactions by themselves are not adequate for structuring long-lived applications [1,2]. One well-known enhancement (supported by the CORBA Object Transaction Service, OTS [3]) is to permit *nesting* of transactions; furthermore, nested transactions could be concurrent. The outermost transaction of

such a hierarchy is typically referred to as the top-level transaction. The durability property is only possessed by the top-level transaction, whereas the commits of nested transactions (subtransactions) are provisional upon the commit/abort of an enclosing transaction. This allows for failure confinement strategies, i.e., the failure of a subtransaction does not necessarily cause the failure of its enclosing transaction. Resources acquired within a subtransaction are inherited (retained) by parent transactions upon the commit of the subtransaction, and (assuming no failures) only released when the top-level transaction completes, i.e., they are retained for the duration of the top-level transaction.

The above enhancement is sufficient if an application function can be represented as a single top-level transaction. Frequently this is not the case. Top-level transactions are most suitably viewed as "short-lived" entities, performing stable state changes to the system [1]; they are less well suited for structuring "long-lived" application functions (e.g., running for hours, days, …). Long-lived top-level transactions may reduce the concurrency in the system to an unacceptable level by holding on to resources for a long time; further, if such a transaction aborts, much valuable work already performed could be undone. In short, if an application is composed as a collection of transactions, then during run time, the entire activity representing the application in execution is frequently required to relax some of the ACID properties of the individual transactions. The entire activity can then be viewed as a non-ACID 'extended transaction'. The spheres of control model [4] describes the underlying concepts of recovery and commitment for extended transactions. Much research work has been done on developing specific extended transaction models [e.g., 5 - 8]. Nevertheless, most of the proposed techniques have not found any widespread usage; indeed, most commercial transaction processing systems do not even support nesting of transactions. One reason cited is lack of flexibility [9], in that the wide range of extended transaction models is indicative that a single model is not sufficient for all applications, so it would be inappropriate to 'hardwire' a specific extension mechanism. In any case, most transaction processing monitors are monolithic in structure, so difficult to extend. Thus the situation remains that a programmer often has to develop application specific mechanisms to build extended transactions.

There is a way out of this situation by exploiting a middleware based approach; in the case of CORBA for example, a set of open services are already available for building distributed applications. Within this context, it is appropriate to examine what additional functionality is required for flexible ways of composing an application using transactions, with the support for enabling the application to possess some or all ACID properties. The CORBA Activity Service Framework described in this paper provides such functionality through a set of structuring mechanisms to complement the OTS.

The design of the service is based on the insight that the various extended transaction models can be supported by providing a general purpose event signalling mechanism that can be programmed to enable activities - application specific units of computations – to coordinate each other in a manner prescribed by the extended transaction model under consideration. This has led to the development of an *Activity Service Framework* which we believe is sufficient to allow middleware to manage

complex business transactions that extend the concept of transaction from the well-understood, short-duration atomic transaction. The different extended transaction models can be mapped onto specific implementations of this framework permitting such transactions to span a network of systems connected indirectly by some distribution infrastructure. The framework described in this paper is an overview the OMG's *Additional Structuring Mechanisms for the OTS* standard [10] now reaching completion. The authors of this paper have been active in all phases this standardisation activity that included defining the scope of the RFP issued in early '99 [11] to making the initial submission and guiding it through to its present form [10]. Although the framework is presented here in CORBA specific terms, the main ideas are sufficiently general, so that it should be possible to use them in conjunction with other middleware.

2. Requirements and Approach

2.1 Requirements

We begin with some examples that illustrate that the need for non-ACID behaviour.

(i) bulletin board: posting and retrieving information from bulletin boards can be performed using transactions. While it is desirable for bulletin board operations to be structured as transactions, if these transactions are nested within other application transactions, then bulletin information can remain inaccessible for long times. Releasing of bulletin board resources early would therefore be desirable. Of course, if the application transaction aborts, it may be necessary to invoke compensating activities; this is consistent with the manner in which bulletin boards are used.

(ii) name server access: Consider the situation where persistent objects have been replicated for availability. The naming service needs to maintain up-to-date information about object replicas to enable clients to be bound to available replicas. For the sake of consistency it is desirable to structure lookup and update operations on the naming service as transactions. Application transactions, upon finding out that certain object replicas are unavailable can invoke operations to update the naming service database accordingly, while carrying on with the main computation [12]. There is no reason to undo these naming service updates should the application transaction subsequently aborts.

(iii) billing and accounting resource usage: if a service is accessed by a transaction and the user of the service is to be charged, then the charging information should not be recovered if the transaction aborts.

These applications share a common feature that as viewed by external users, in the event of successful execution (i.e., no machine failures or application-level exceptional responses which force transactions to rollback), the work performed possesses all ACID features of traditional transactional applications. If failures occur, however, non-ACID behaviour is possible, typically resulting in non-serializability. For some applications, e.g., the name service example above, this does not result in application-level inconsistency, and no form of compensation for the failure is

required. However, for other applications, e.g., the bulletin board, some form of compensation may be required to restore the system to a consistent state from which it can then continue to operate.

(iv) Long-running business activity: Long-running activities can be structured as many independent, short-duration top-level transactions, to form a "logical" long-running transaction. This structuring allows an activity to acquire and use resources for only the required duration of this long-running transactional activity. This is illustrated in fig. 1, where an application activity (shown by the dotted ellipse) has been split into many different, coordinated, short-duration top-level transactions. Assume that the application activity is concerned with booking a taxi (*t1*), reserving a table at a restaurant (*t2*), reserving a seat at the theatre (*t3*), and then booking a room at a hotel (*t4*), and so on. If all of these operations were performed as a single transaction then resources acquired during *t1* would not be released until the top-level transaction has terminated. If subsequent activities *t2*, *t3* etc. do not require those resources, then they will be needlessly unavailable to other clients.

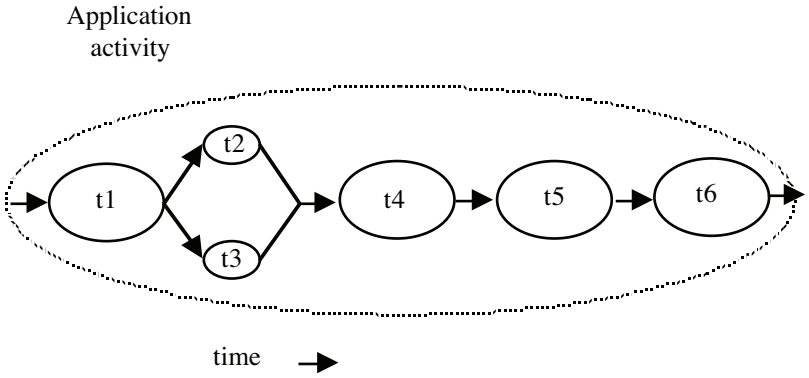

Figure 1: An example of a logical long-running "transaction", without failure.

However, if failures and concurrent access occur during the lifetime of these individual transactional activities then the behaviour of the entire "logical long-running transaction" may not possess ACID properties. Therefore, some form of (application specific) compensation may be required to attempt to return the state of the system to (application specific) consistency. For example, let us assume that *t4* aborts (fig. 2). Further assume that the application can continue to make forward progress, but in order to do so must now undo some state changes made prior to the start of *t4* (by *t1*, *t2* or *t3*). Therefore, new activities are started; *tc1* which is a compensation activity that will attempt to undo state changes performed, by say *t2*, and *t3* which will continue the application once *tc1* has completed. *tc5'* and *tc6'* are new activities that continue after compensation, e.g., since it was not possible to reserve the theatre, restaurant and hotel, it is decided to book tickets at the cinema. Obviously other forms of transaction composition are possible.

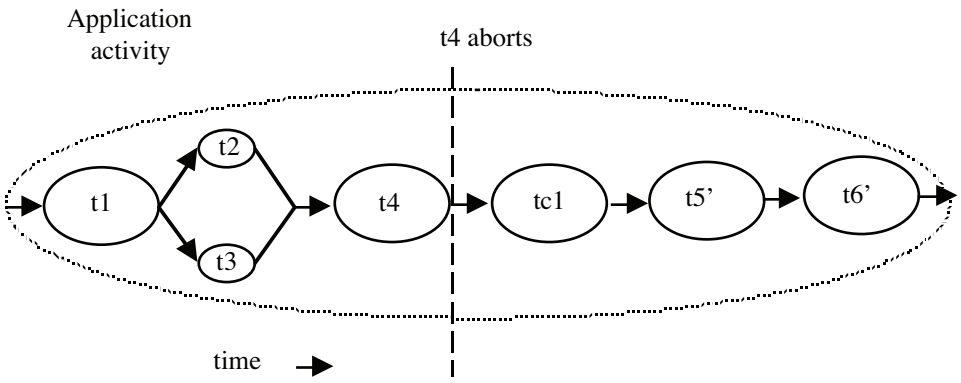

Figure 2: An example of a logical long-running "transaction", with failure.

There are several ways in which some or all of the application requirements outlined above could be met. However, it is unrealistic to believe that the "one-size fits all" paradigm will suffice, i.e., a single approach to extended transactions is unlikely to be sufficient for all (or even the majority of) applications. Whereas in case of the last example, a transactional workflow system with scripting facilities for expressing the composition of the activity with compensation (a workflow) may be the most suitable approach, a less elaborate solution might be desirable for the first three examples.

2.2 Approach

As hinted earlier, the approach taken in the CORBA Activity Service is to provide a low-level infrastructure capable of supporting the coordination and control of abstract, application specific entities to enable construction of various forms of extended transaction models as desired by workflow engines, component management middleware and other systems. As we shall see, these entities (*activities*) may be transactional, they may use weaker forms of serializability, or they may not be transactional at all. The important point is that a computation is viewed as composed of one or more activities and the activity service is only concerned with their control and co-ordination, leaving the semantics of such activities to the application programmer. As is the case with other middleware standards, the Activity Service does not specify the implementation details of how the activities should be coordinated, only providing interfaces for coordination to occur.

An activity containing component activities may impose a requirement on the Activity Service implementation for managing these component activities. It must be determined whether these component activities worked as specified or failed or terminated exceptionally and how to map their completion (or non-completion) to the enclosing activity's outcome. This is true whether the activities are strictly parallel, strictly sequential or some combination of the two. In general, an activity (or some entity acting on its behalf) that needs to co-ordinate the outcomes of component activities has to know what state each component activity is in:

which are active

which have completed and what their outcomes were

which activities failed to complete

This knowledge needs to be related to its own eventual outcome. A responsible entity may be required to handle the sub-activity outcomes, and this can be modelled as an (distinguished) activity so that control flows can be made explicit. The activity determines the collective outcome of the component activity in the light of the various failure and success situations its component activities present it with.

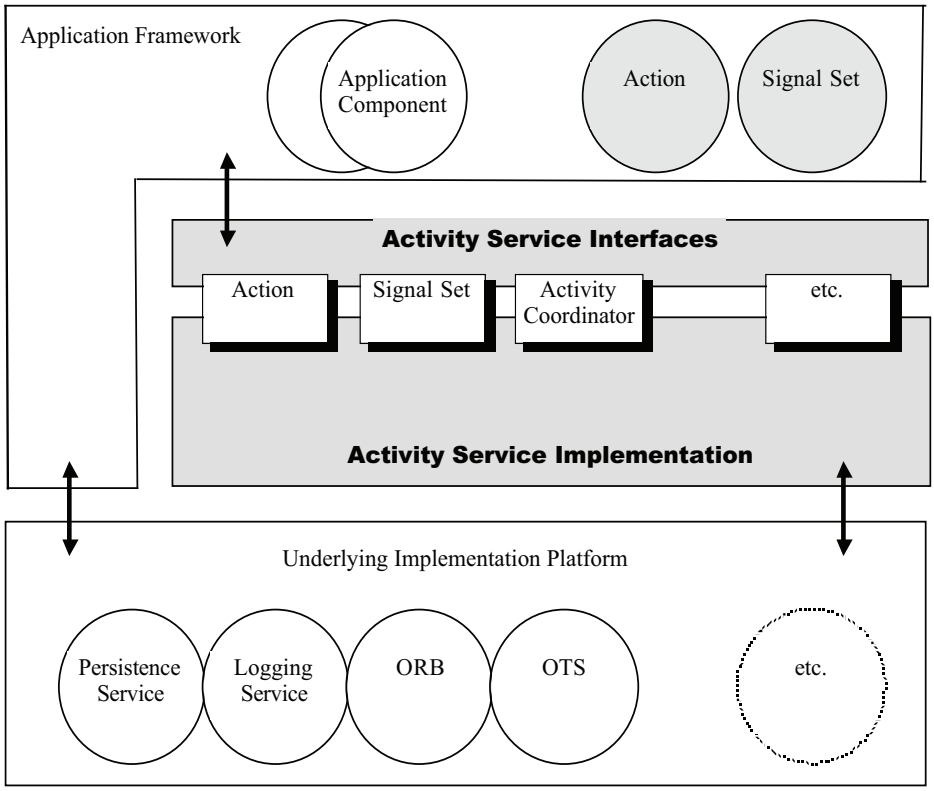

Figure 3: The role of the Activity Service.

The activity service meets the above requirements in a very simple manner. Basically, associated with each activity is an *activity coordinator* that can coordinate the execution of constituent activities. In general, the coordination required can vary depending upon the phase of the execution of the entity (e.g., starting, terminating), so associated with a coordinator are one or more *signal sets*, each such set implementing a specific coordination protocol. For example, a signal set could implement a two phase commit protocol. Constituent activities are required to register themselves with a given signal set of the coordinating activity; this is done by an activity registering an *action* with the signal set. At an appropriate time, the coordinating activity triggers the execution protocol implemented by one of its signal set by invoking a standard operation; this leads to the set *signalling* each registered activity by invoking an operation on the registered action. The signalled activity can now perform some specific computation and return results (e.g., flush the data on to stable store and return 'done'), and this way the protocol advances. These aspects of activity coordination are discussed at length in the subsequent sections.

A very high level view of the role of the Activity Service is shown in fig. 3. It is not expected that the operations in the Activity Services interfaces will be used directly by end-user application programmers. When we talk about application programmers

here we mean those who write for example, application framework for workflow managers or component management systems or who are extending the functionality of the Containers of Enterprise Java Beans (EJBs).

3. The Activity Service framework

3.1 Activities

An *activity* is a unit of (distributed) work that may, or may not be transactional. During its lifetime an activity may have transactional and non-transactional periods. Every entity including other activities can be parts of an activity, although an activity need not be composed of other activities. Each activity is represented by an *activity object*. An activity is *created*, made to *run*, and then *completed*. The result of a completed activity is its *outcome*, which can be used to determine subsequent flow of control to other activities. Activities can run over long periods of time and can thus be *suspended* and then *resumed* later.

Demarcation signals of any kind are communicated to registered entities (*actions*) through *signals*. For example, the termination of one activity may initiate the start/restart of other activities in a workflow-like environment. Signals can be used to infer a flow of control during the execution of an application. Importantly, signals may be communicated at arbitrary points during the lifetime of an activity and not just when it terminates.

One of the keys to the extensibility of this framework is the *signal set* whose implemented behaviour is peculiar to the kind of extended transaction. The signal set is the entity that generates signals that are sent to actions and processes the results returned to determine which signal to send next. Similarly, the behaviour of an action will be peculiar to the extended transaction model of which it is a part. So as new types of extended transaction models emerge, so will new signal set instances and associated actions. This allows a single implementation of this framework to serve a large variety of extended transaction models, each with its own idea of extended transactions, each with its own action and signal set implementations. The *Activity Service implementation* will not need to know the behaviour which is encapsulated in the actions and signal sets it is given, merely interacting with their interfaces in an entirely uniform and transparent way.

3.2 Activity coordination and control

An activity may run for an arbitrary length of time, and may use atomic transactions at arbitrary points during its lifetime. For example, consider fig. 4, which shows a series of connected activities co-operating during the lifetime of an application. The solid ellipses represent transaction boundaries, whereas the dotted ellipses are activity boundaries. Activity *A1* uses two top-level transactions during its execution, whereas *A2* uses none. Additionally, transactional activity *A3* has another transactional activity, *A3'* nested within it.

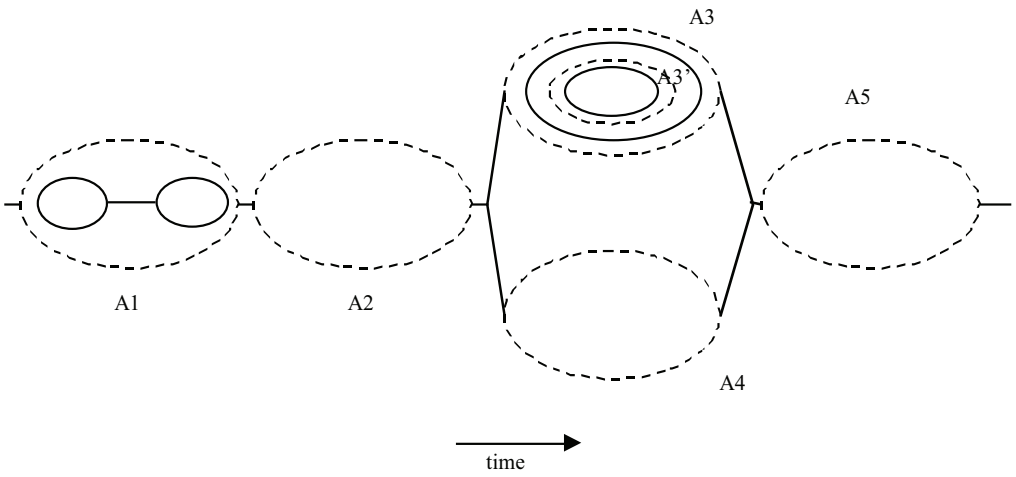

Figure 4: Activity and transaction relationship.

3.2.1 Completion status

When an Activity completes, it does so in one of two states, either *success* or *failure*. During its lifetime, the completion state of the Activity (i.e., the state it would have if it completed at that point) may change from success to failure, and back again many times. This is represented by the CompletionStatus enumeration, whose values are:

CompletionStatusSuccess: the Activity has successfully performed its work and can complete accordingly. When in this state, the Activity completion status can be changed.

CompletionStatusFail: some (application specific) error has occurred which has meant that the Activity has not performed all of its work, and should be driven during completion accordingly. When in this state, the Activity completion status can be changed.

CompletionStatusFailOnly: some (application specific) error has occurred which has meant that the Activity has not performed all of its work, and should be driven during completion accordingly. Once in this state, the completion status of the Activity cannot be changed, i.e., the only possible outcome for the Activity is for it to fail.

The interpretation of the completion status outcome to drive specific Signals and Activity specific work is up to the actual Activity.

3.2.2 Actions and Signals

An activity may decide to transmit activity specific data (*Signals*) to any number of other activities at specific times during its lifetime, e.g., when it terminates. The receiving activities may either have been running and are waiting for a specific Signal, or may be started by the receipt of the Signals. The information encoded within a Signal will depend upon the implementation of the extended transaction model and therefore the definition of the Signal is designed to accommodate this.

```
struct Signal
{
    string signal_name;
    string signal_set_name;
    any    application_specific_data;
};
```

To allow activities to be independent of the other activities, and also to allow the insertion of arbitrary coordination and control points, Signals are sent to *Actions*. An Action can then use the Signal in an application specific manner and return an indication of it having done so.

```
interface Action
{
    Outcome process_signal(in Signal sig) raises(ActionError);
};
```

3.2.3 SignalSets

To drive the Signal and Action interactions an *activity coordinator* is associated with each activity. Activities that require to be informed when another activity sends a specific Signal can register an appropriate Action with that activity's coordinator. When the activity sends a Signal (e.g., at termination time), the coordinator's role is to forward this signal to all registered Actions and to deal with the outcomes generated by the Actions.

The implementation of the coordinator will depend upon the type of extended transaction model being used. For example, if a Sagas type model [6] is in use then a compensation Signal may be required to be sent to Actions if a failure has happened, whereas a coordinator for a CA action model [13] may be required to send a Signal informing participants to perform exception resolution. Therefore, to enable the coordinator to be configurable for different transaction models, the coordinator delegates all Signal control to the SignalSet. Signals are associated with SignalSets and it is the SignalSet that generates the Signals the coordinator passes to each Action. The set of Signals a given SignalSet can generate may change from one use to another, for example based upon the current status of the Activity or the responses from Actions. The intelligence about which Signal to send to an Action is hidden within a SignalSet and may be as complex or as simple as is required. Importantly, a SignalSet is dynamically associated with an activity, and each activity can have a different SignalSet controlling it.

```
interface SignalSet
{
    readonly attribute string signal_set_name;

    Signal get_signal (inout boolean lastSignal);
    Outcome get_outcome () raises(SignalSetActive);

    boolean set_response (in Outcome response,
                          out boolean nextSignal)
                                    raises (SignalSetInactive);

    void set_completion_status (in CompletionStatus cs);
    CompletionStatus get_completion_status ();
};
```

The activity coordinator therefore interacts with the SignalSet to obtain the Signal to send to registered Actions, and passes the results back to the SignalSet, which can collate them into a single result (fig. 5). Which SignalSet is used by the coordinator will depend upon factors such as the type of extended transaction model being used or the state of the activity (e.g., rolling back or committing), that is indicated by an appropriate CompletionStatus value.

When a Signal is sent to an Action, the SignalSet is informed of the result generated by that Action to receiving and acting upon that Signal; the SignalSet may then use that information when determining the nature of the next Signal to send. When a given Signal has been sent to all registered Actions the SignalSet will be asked by the coordinator for the next Signal to send.

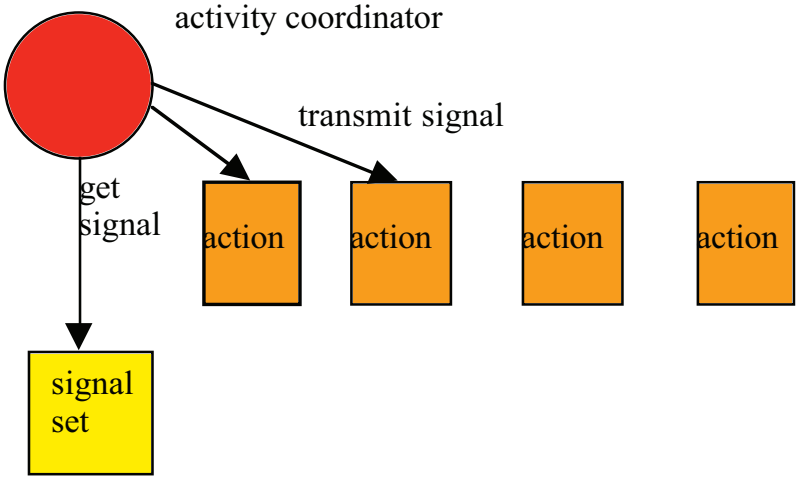

Figure 5: Activity coordinator signalling actions.

Since it may not be possible to determine beforehand the set of Signals that will be generated by a SignalSet, Actions register interest in SignalSets, rather than specific Signals. Whenever a SignalSet generates any Signal, those Actions which have registered interest in that SignalSet will receive the Signal. An Action may register

interest in more than one SignalSet and an activity may use more than one SignalSet during its lifetime (fig. 6).

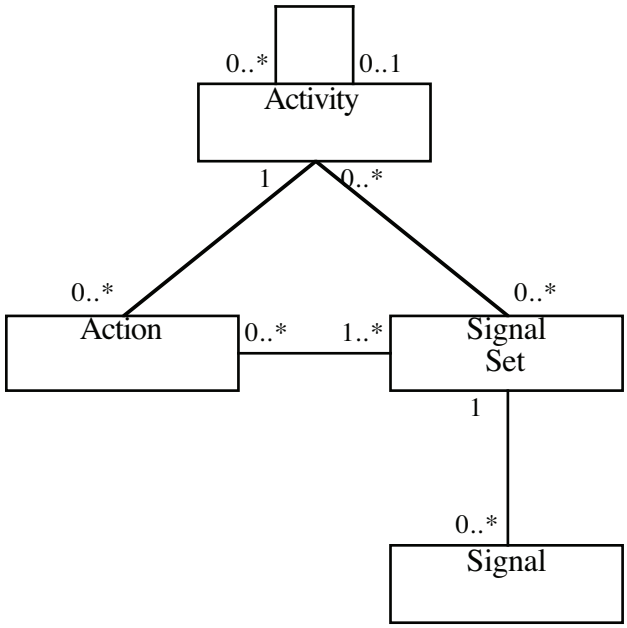

Figure 6: Relationship of SignalSets, Signals, Actions and Activities.

As shown in fig. 7, a given SignalSet is assumed to implement a state machine, whereby it starts off in the *Waiting state* until it is required by the Activity Coordinator to send its first Signal, when it then either enters the *Get Signal state* or the *End state* if it has no Signals to send. Once in the *End state* the SignalSet cannot provide any further Signals and will not be reused. Once in the *Get Signal* state the SignalSet will be asked for a new Signal until it enters the *End state*. A new Signal is only requested from the SignalSet when all registered Actions have been sent the current Signal.

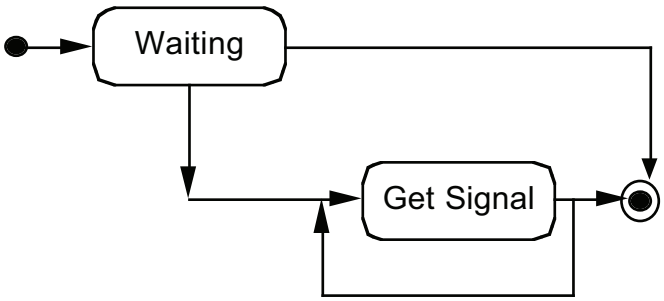

Figure 7: SignalSet state transition diagram.

With the exception of some predefined Signals and SignalSets, the majority of Signals and SignalSets will be defined and provided by the higher-level applications that make use of this Activity Service framework. To use the generic framework provided within this specification it is necessary for these higher-level applications to impose application specific meanings upon Signals and SignalSets, i.e., to impose a structure on their abstract form. Illustrative examples are given in section 4.

3.3 Properties

The programmer possesses application specific knowledge about how the application will use data, e.g., how locks on data should be obtained, and how activities should deal with failures. An encompassed activity that needed to perform an update could override this. This configuration information may change during the lifetime of the application, as users requirements change. If such information were hard-wired into the application, each time a change to the configuration is made, the application would have to be rebuilt.

Therefore, what is required is a way to store this information as data, which can be modified without requiring changes to the applications and activities that use it. In addition, such data may be required to be shared between distributed activities. However, how this data is stored and accessed may also depend upon the application requirements. Therefore, rather than mandate a specific implementation for managing such properties, we simply provide a mechanism for applications to obtain their own "property store" implementations. This is the role of the *PropertyGroup*. A PropertyGroup represents properties as a tuple-space of attribute-value pairs.

A PropertyGroup may be associated with each (distributed) Activity. A PropertyGroup manages a group of properties and defines their behaviour with respect to:

the visibility of changes made to properties in a nested Activity.

the visibility of changes made to properties in "downstream" nodes.

the manner in which property values are accessed in "downstream" nodes, i.e., whether properties are propagated by value or by reference.

An Activity can support any number of registered PropertyGroups, each with its own set of behaviour. Different PropertyGroup implementations may have different behaviours with respect to nested Activities. For example, one type of PropertyGroup may allow updated properties to be transmitted within nested contexts, while another may not. There are obviously scenarios where both types of PropertyGroup could be used at the same time, e.g., PG1 could represent "client environment" information such as locale or codepage; overriding of this information within nested contexts would make no sense; PG2 may represent application context, certain parts of which may require to be available only for the specific context in which they were set.

3.4 Treatment of failure and recovery

The failure of an individual activity may produce application specific inconsistencies depending upon the type of activity.

if the activity was involved within a transaction, then any state changes it may have been making when the failure occurred will eventually be recovered automatically by the transaction service.

if the activity was not involved within a transaction, then application specific compensation may be required.

an application that consisted of the (possibly parallel) execution of many activities (transactional or not) may still require some form of compensation to "recover" committed state changes made by prior activities. For example, the application shown in fig. 2.

Rather than distinguish between compensating and non-compensating activities, we consider that the compensation of the state changes made by an activity is simply the role of another activity. A compensating activity is simply performing further work on behalf of the application. Just as application programmers are expected to write "normal" activities, they will therefore also be required to write "compensating" activities, if such are needed. In general, it is only application programmers who possess sufficient information about the role of data within the application and how it has been manipulated over time to be able to compensate for the failure of activities. For example, suitable Actions may be created that compensate for work performed by an Activity, and triggered only if a specific SignalSet is used (see the example given in section 4.2).

Recovering applications after failures, such as machine crashes or network partitions, is an inherently complex problem: the states of objects in use prior to the failure may be corrupt, and the references to objects held by remote clients may be invalid. At a minimum, restoring an application after a failure may require making object states consistent. The advantage of using transactions to control operations on persistent objects is that the transactions ensure the consistency of the objects, regardless of whether or not failures occur.

Rather than mandate a particular means by which objects should make themselves persistent, many transaction systems simply state the requirements they place on such objects if they are to be made recoverable, and leave it up to the object implementers to determine the best strategy for their object's persistence. The transaction system itself will have to make sufficient information persistent such that, in the event of a failure and subsequent recovery, it can tell these objects whether to commit any state changes or roll them back. However, it is typically not responsible for the application object's persistence.

In a similar way, we only state what the requirements are on such a service in the event of a failure, and leave it to individual implementers to determine their own recovery mechanisms. Unlike in a traditional transactional system, where crash recovery mechanisms are responsible for guaranteeing consistency of object data, the types of extended transaction applications we envision using this service will typically also require the ability to recover the activity structure that was present at the time of the failure. This will then enable the activity application to then progress onwards. However, it is not possible for the Activity Service to perform such complete recovery on its own; it will require the co-operation of the Transaction Service, the Activity Service and the application. Since it is the application logic that imposes meaning on Actions, Signals, and SignalSets in order to drive the activities to completion during normal (non-failure) execution, it is predominately this logic that is required to drive recovery and ensure activity components become consistent.

The recovery requirements imposed on the Activity Service and the applications that use it can be itemised as follows:

application logic: the logic required to drive the activities during normal runtime will be required during recovery in order to drive any in-flight activities to application specific consistency. Since it is the application level that imposes meaning on Actions, Signals, and SignalSets, it is predominately the application that is responsible for driving recovery.

rebinding of the activity structure: any references to objects within the activity structure which existed prior to the failure must be made valid after recovery.

application object consistency: the states of all application objects must be returned to some form of application specific consistency after a failure.

recover actions and signal sets: any Actions and SignalSets used to drive the activity application must be recovered.

Finally, a few words on the delivery of Signals. Minimally, the delivery semantics for Signals is required to be *at least once*, although implementations are free to provide better deliver guarantees. This means that an Action may receive the same Signal from an Activity multiple times, and must ensure that such invocations are idempotent, i.e., that multiple invocations of the same Signal to an Action are the same as a single invocation. Stronger delivery semantics - *exactly once* – can be provided by the activity service itself making use of the underlying transaction service.

4. Examples

In this section we describe how the Activity Service can be used to support a variety of coordination protocols, ranging from two-phase commit, workflow coordination to coordination of Web Services.

4.1 Two-phase commit

We begin with a simple example illustrating how the Activity Service can be used to implement the classic transaction commit protocol; fig. 8 shows the exchanges involved when the transaction commits. The coordinating activity initiates commit by invoking get_signal operation of its 2PCSignalSet. The Set returns a ‘prepare’ signal that is sent to the first registered Action, whose response – done, rather than abort in this case - is communicated to the Set (operation set_response); the Set returns the prepare signal again that is then sent to the next registered Action and so forth.

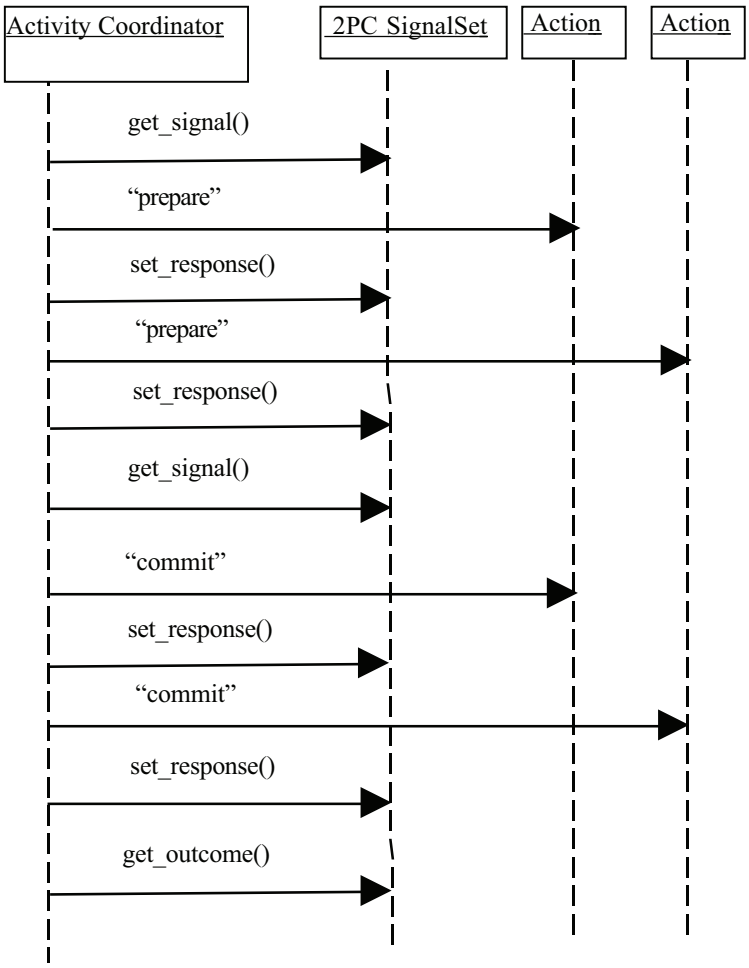

Figure 8: Two-phase commit protocol with Signals, SignalSets and Actions.

4.2 Nested top-level transactions with compensations

We next illustrate how coordination of transactional activities with compensation for failures can be provided using the framework described. Consider the sequence of transactions shown in fig. 9, and assume as before that solid ellipses represent transaction boundaries and dotted ellipses represents an enclosing activity.

What we want to provide is the situation where within a top-level transaction (A), the application can start a new top-level transaction (B) that can commit or rollback independently of A. This scheme (also called open nested transactions [8]) can be useful if resources are required for only a short duration of the transaction A (as in the bulletin board example, section 2.1). If A subsequently commits then there is no problem with application consistency. However, if A rolls back, then it is possible that the work performed by B may be required to be undone (represented by transaction !B).

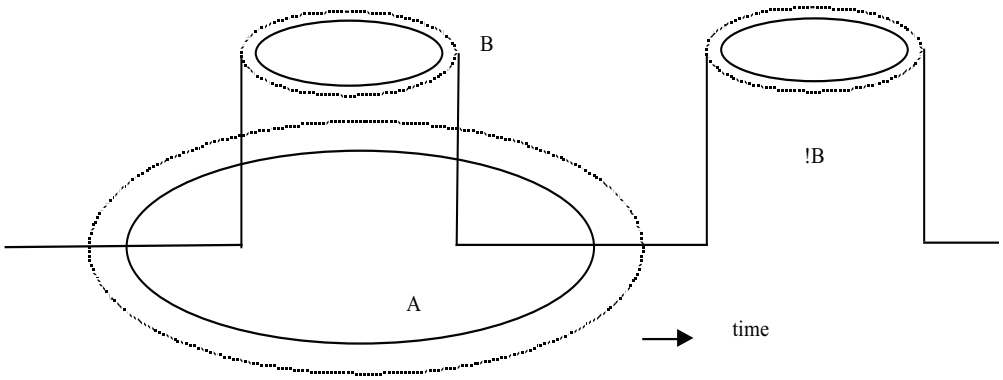

Figure 9: Nested top-level transactions.

We make the following assumptions: (i) that each enclosing activity has a single SignalSet that is used when the activity completes (say, the CompletionSignalSet), and this SignalSet has Success, Failure and Propagate Signals, depending upon whether it completes successfully (and has no dependencies on other activities), completes abnormally (aborts), or completes successfully but has other activity dependencies, respectively; (ii) there is an Action that is responsible for starting !B if it receives the Failure Signal from an enclosing activity (say, the CompensationAction); the "state transitions" for the Action are:

If it receives the Success Signal then it can remove itself from the system.

If it receives the Propagate Signal, then encoded within this Signal will be the identity of an Activity it should register itself with. It must also remember that it has been propagated.

If it receives the Failure Signal and it has never been propagated then it can remove itself from the system. If the Action has been propagated then it should start !B running, before removing itself.

Then the above structure can be obtained in the following manner:

When transaction A's enclosing activity is begun, it registers the CompletionSignalSet as the one to use when the activity terminates. At this point no Actions are registered with that activity and hence with the SignalSet.

When B is begun (and hence it's enclosing activity is also started), the activity registers the CompensationAction with B's activity, i.e., it's CompletionSignalSet.

If B commits, the enclosing activity will terminate in the successful state, and the CompletionSignalSet will have the coordinator send the Propagate Signal to the registered CompensationAction. Encoded within this Signal will be the identity of the activity to propagate to, i.e., A. The CompensationAction can then enlist itself with A.

If B rolls back, the enclosing activity will terminate in the failure state, and the CompensationAction will do nothing when it receives the Failure Signal.

If A subsequently commits, it's enclosing activity's CompletionSignalSet will generate the Success Signal (since it has no dependencies on other activities), which will be delivered to the CompensationAction. In this case, no compensation is required, so the Action does nothing.

On the other hand, if A subsequently rolls back, it's enclosing activity's CompletionSignalSet will generate the Failure Signal, and the CompensationAction will start !B to undo B.

4.3 LRUOW: Long Running Unit Of Work

The LRUOW model described in [14] is another extended transaction model to support long-running transactions. It combines some of the semantics of nested transactions and type-specific concurrency control; it relies on being able to execute long-running transactions in two phases: the *rehearsal phase*, where the work is performed without recourse to serializability and which may take an arbitrary amount of time and the *performance* phase, where the work is confirmed (committed) only if suitable locks and consistency criteria can be obtained on the data. In order to do this, it is necessary to have sufficient support from the resources used within the transactions, and to be able to specify operation predicates.

The LRUOW model could be implemented on the activity service infrastructure using a Rehearsal SignalSet and a Performance SignalSet. Each LRUOW resource could register a suitable Action with each SignalSet which would be driven when the activity completes. The higher-level API proposed in [14] would still be applicable, but would be mapped down to using these SignalSets and Actions. Each transaction would also be enclosed within an activity, which would be responsible for propagating resources from the child to the parent if the transaction completes successfully. This has the advantage that no modification to existing transaction systems would be required.

4.4 Workflow coordination

Transactional workflow systems with scripting facilities for expressing the composition of an activity (a business process) offer a flexible way of building application specific extended transactions. Here we describe how the Activity Service Framework can be utilised for coordinating workflow activities. The signal set required to coordinate a business activity contains four signals, "start", "start_ack", "outcome" and "outcome_ack".

start: signal is sent from a "parent" activity to a "child" activity, to indicate that the "child" activity should start. The *application_specific_data* part of the signal contains the information required to parameterise the starting of the activity.

start_ack: signal is sent from a "child" activity to a "parent" activity, as the return part of a "start" signal, to acknowledge that the "child" activity has started.

outcome: signal is sent from a "child" activity to a "parent" activity, to indicate that the "child" activity has completed. The *application_specific_data* part of the

signal contains the information about the outcome of the activity, e.g., whether or not it completed successfully.

outcome_ack: signal is sent from a "parent" activity to a "child" activity, as the return part of an "outcome" signal, to acknowledge that the "parent" activity has completed.

The interaction depicted in fig. 10 is activity *a* coordinating the parallel execution of *b* and *c* followed by *d.*

Referring to fig. 1 we can tie an activity to a single top-level transaction, such that when an activity begins (e.g., t1) it immediately starts a new transaction. A coordinating activity (implied by the dotted ellipse in the figure) would send appropriate "start" Signals, and wait for the "outcome" Signals to occur.

To do this, each potential activity registers an Action with a specific SignalSet at the coordinating activity (the parent); each activity that needs to be started for a specific event would register an Action with a specific SignalSet, e.g., t2 and t3 would register with the same SignalSet since they need to be started together, whereas t4 would be registered with a separate SignalSet.

Whenever a child activity is started the parent activity registers an Action with it that is used to deliver the "outcome" Signal to the parent. Let's assume that each child activity has a Completed SignalSet to facilitate this. When a child activity terminates, it uses the Completed SignalSet to send a Signal to the parent's registered Action. The content of this Signal will contain sufficient information for the parent to determine the outcome of the activity, and use this to control the flow of activities appropriately.

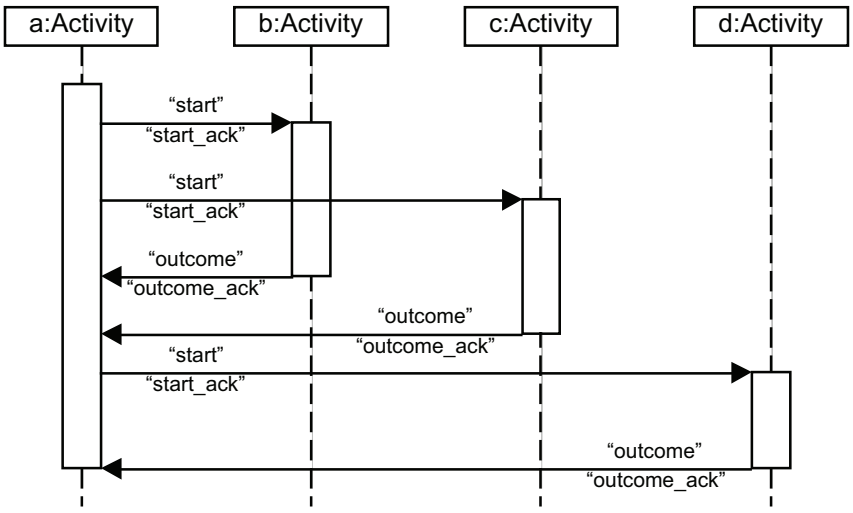

Figure 10: Workflow coordination.

For example, in fig. 2, the parent activity would receive a successful termination outcome from t1, which would cause it to send "start" Signals to t2 and t3 via their registered Actions. When they both complete successfully (i.e., sent "outcome" Signals), it can then start t4. However, if t4 sends a failure outcome, or simply fails to send any outcome (e.g., it crashes), the parent activity can use this information to start tc1 in order to do the compensation.

The task (i.e., activity) coordination scheme used in the OPENflow transactional workflow management system [15] is very similar to the above scheme. Here, associated with each task is a transactional *task controller* object. The purpose of a task controller is to receive notifications of outputs of other task controllers and use this information to determine when its associated task can be started. The task controller is also responsible for propagating notifications of outputs of its task to other interested task controllers.

4.5 Business Transaction Protocol (BTP)

There has been much work recently on enriching Web Services with transactional properties. Assume that the long running business activity (see fig. 1) represents the activity of a composite Web Service built out of individual Web Services provided by different organisations. As explained in section 2, such an activity needs to be managed as an extended transaction. The Business Transaction Protocol (BTP) is one such extended transaction protocol defined by the Organization for Advance Structured Information Systems (OASIS) [16]. BTP is designed to support applications which are disparate in time, location, and administration and thus require transactional support beyond classical ACID transactions.

BTP defines two types of 'transactions', which we shall briefly outline:

atoms, which execute a traditional two-phase commit protocol on all the enlisted participants. Unlike ACID transactions, there are no implied semantics about how the protocol is implemented (and enforced) by participants. So, for example, two-phase locking of resources is not a requirement. In addition, users are expected to drive both phases of the protocol explicitly, i.e., issue prepare followed (at an arbitrary time later) by either confirm or cancel (BTP uses *confirm* in place of *commit* and *cancel* for *rollback)*. Individual services (participants) are free to implement prepare, confirm and cancel in a manner appropriate to them. Issues to do with consistency and isolation of data are also matters private to individual services and *not* imposed or assumed by BTP.

cohesions are non-ACID transactions and allow for the selection of work to be confirmed or cancelled based on higher level business rules. Atoms are the typical participants within a cohesion but, unlike an atom, a cohesion may give different outcomes to its participants such that some of them may confirm whilst the remainder cancel. In essence, the two-phase protocol for a cohesion is parameterised to allow a user to specify precisely which participants to prepare and which to cancel. The strategy underpinning cohesions is that they better model long-running business activities, where services enroll in atoms that represent specific units of work and as the business activity progresses, it may encounter conditions that allow it to cancel or prepare these units, with the caveat that it may be many hours or days before the cohesion arrives at its *confirm-set*: the set of participants that it requires to confirm in order for it to successfully terminate the business activity. Once the confirm-set has been determined, the cohesion collapses down to being an atom: all members of the confirm-set see the same outcome.

Providing an implementation of atoms is straightforward: there are two SignalSets with which all participants are registered: the PrepareSignalSet and the

CompleteSignalSet. As shown in fig. 11, a user invokes the prepare phase of the atom protocol by causing the ActivityCoordinator to drive the PrepareSignalSet, which sends the *prepare* Signal to all Actions. A user can obtain the final result of this stage of the protocol via get_outcome.

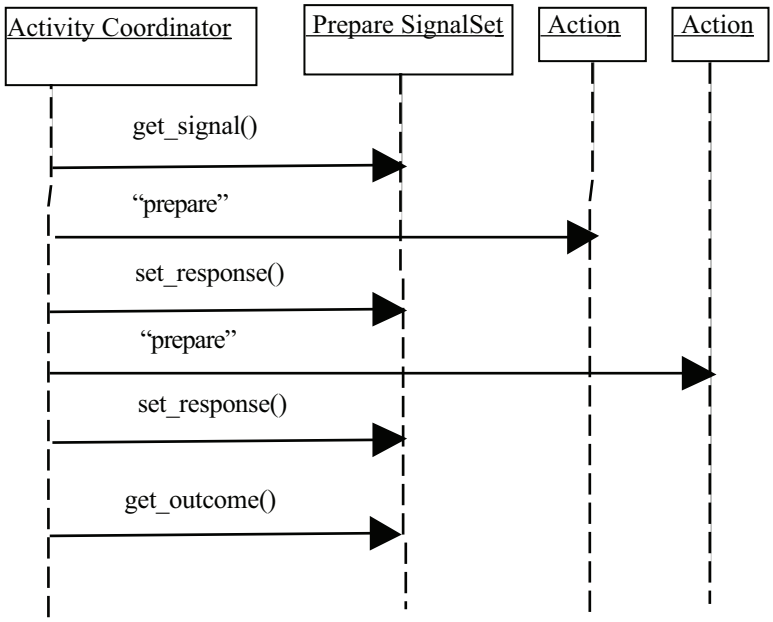

Figure 11: The BTP PrepareSignalSet.

At this stage, all Actions are prepared but the choice as to whether to confirm or cancel the atom is up to the user and not the coordinator, as it is in the case of traditional ACID transactions. The CompleteSignalSet can either issue a *confirm* or a *cancel* Signal, depending upon how the atom is instructed to terminate. Assuming that the atom is to confirm (indicated to the ActivityService by a success CompletionStatus), figure 12 shows the Signal processing:

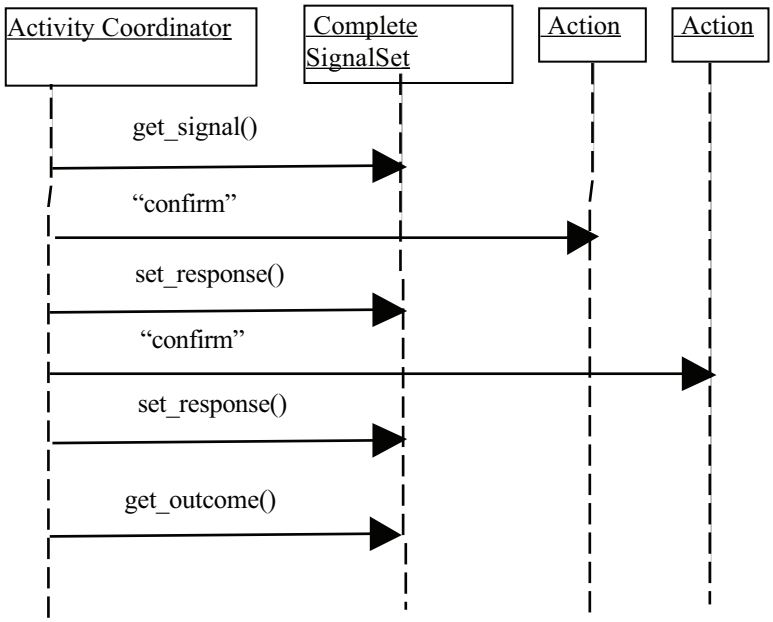

Figure 12: The BTP CompleteSignalSet.

If the atom is instructed to cancel, then obviously the *confirm* Signal is replaced by *cancel*.

It is obviously possible to implement cohesions in a similar manner, although with more effort since the participants in the two phases of the completion protocol need not be the same, as they are with atoms. Referring to fig. 1, the dotted ellipse would be the cohesion which will manage the overall (enclosed) business interactions. Each of the enclosed ellipses represents a separate atom and the end of an ellipse is just the preparatory phase and not the actual termination, e.g., for t1 the taxi is reserved (prepared) and not booked (confirmed): that is the role of the cohesion termination protocol.

So, the business logic creates atoms and enrols them with cohesions before invoking services within the scope of those atoms (e.g., taxi, theatre reservation etc.). If all of the services can be reserved successfully, then the application can decide to terminate the cohesion (the end of the dotted ellipse). At this point, the business logic has obtained its confirmation set and the cohesion confirmation semantics mean that it guarantees atomicity across all members of the set, i.e., all atoms (and hence all work performed with those atoms) will be confirmed or none will.

Now consider the case shown in fig. 2 where atom t4, the hotel reservation, fails or the price quoted does not match the users acceptance criteria. Therefore, either implicitly due to failure, or explicitly due to the business logic, t4 is cancelled. In this case, although the hotel reservation atom may be cancelled, it is possible that some work has been performed. Hence the cancellation atom is required and also enrolled with the cohesion. Assuming the hotel reservation can be undone (i.e., tc1 prepares successfully), the application knows that it is safe to execute the remaining atoms and finally arrive at the new confirmation set.

5. Future directions

Since the initial development of the CORBA Activity Service, there has been much interest and continued work on its development and use in CORBA and other environments. In this section we shall outline some of those developments.

5.1 The J2EE Activity Service

J2EE is Java-based enterprise distributed computing platform. As with CORBA, it is essentially a collection of standards, developed by industry committees. Many of the standards are either simply Java language mappings of the CORBA equivalents or are based on CORBA standards. For example, the Java Transaction Service is a Java mapping of the OTS.

Distributed and local transactions play a very important part in any enterprise middleware and in J2EE particularly. For example, in JDBC (database connectivity), distributed two-phase transactions are required when accessing databases and local transactions are mandated within the Java Messaging Service (distributed transactions are optional) [18]. However, all of these transactions are ACID and J2EE is now being enriched to incorporate the activity service from the world of CORBA.

At the time of writing (August 2002), the J2EE Activity Service specification (JSR 95) is still under development [19]. It's aims are similar to JTA, the J2EE Java Transaction API [20]: to provide a higher-level API within the J2EE architecture that simplifies the use of the Activity Service. As with the JTA, the J2EE Activity Service does not mandate that a CORBA implementation lies underneath the APIs it provides, but does require that, for interoperability purposes, format of distributed invocations are compliant with their CORBA equivalents.

A high-level overview of the architecture is shown in fig. 13.

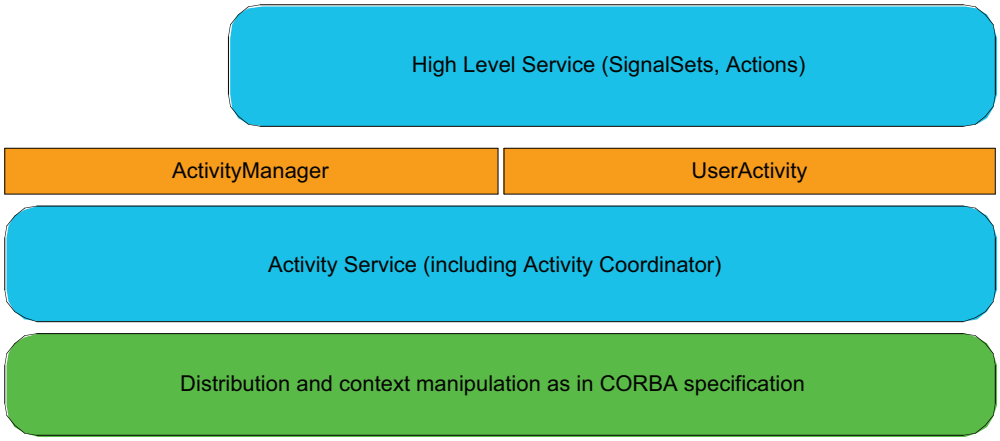

Figure 13: J2EE Activity Service.

The high-level service (HLS) specifies a specific extended transaction model. As such, it is the responsibility of the HLS implementer to provide appropriate SignalSets and specify the associated protocol that Action implementations use (via the related Signals and Outcomes). The HLS relies on the Activity Service to manage the context distribution and relationships between Activities and any transactions.

The ActivityManager provides a simplified the way in which HLS implementers interact with the underlying Activity Service implementation. The implementations the HLS needs to provide in order to configure the Activity Service (e.g., the SignalSet) can be plugged into the underlying implementation via appropriate methods. Activities can be demarcated through UserActivity.

There is also a separate effort going on for standardising and using the PropertyGroup concept provided by the Activity Service [17].

5.2 Web Services Coordination Framework

In the previous section we described the BTP model of Web Services transactions. However, BTP is just another example of a specific extended transaction model. It is likely that other extended transaction models may be required for Web Services. The Activity Service can be used as a basis of supporting a family of extended transaction models for Web Services. In this connection, the Web Services Coordination Framework (WSCF) currently under development within industry [21] is worth noticing.

Although the framework is intended to be a core part of the Web Services architecture for general coordination, probably the first two uses to which it will be put are transactional: (i) providing ACID transaction support and (ii) to support BTP. We have already shown how both of these may be supported. Although WSCF is yet to be finalised, the only noticeable difference between the Web Services version of the Activity Service and its CORBA original, is that the former does not assume an underlying OTS implementation: all coordination services (including transactions) must be constructed on top of the framework.

6. Concluding remarks

We have presented the core elements of the Activity Service Framework that is described in detail in the OMG specification [11]. Although it has long been realised that ACID transactions by themselves are not adequate for structuring long-lived applications and much research work has been done on developing specific extended transaction models, no middleware support for building extended transactions is currently available and the situation remains that a programmer often has to develop application specific mechanisms. The CORBA Activity Service Framework described in this paper is a way out of this situation; it provides a general purpose event signalling mechanism that can be programmed to enable activities to coordinate each other in a manner prescribed by the model under consideration. Through a number of examples we have shown that the Framework has the flexibility to support the coordination required by a wide variety of extended transaction models, ranging from two-phase commit, workflow coordination to the coordination of Web Services.

7. Acknowledgements

Discussions with the partners involved in the development of the Standard is gratefully acknowledged; these include: Eric Newcomer (IONA Technologies), Malik Saheb (INRIA), Michel Ruffin (Alcatel), Shahzad Aslam-Mir (VERTEL/Expersoft) and Nhan T. Nguyen (Bank of America). The work at the Newcastle University was supported in part by a grant from IBM, Hursley.

References

[1] J. N. Gray, “The transaction concept: virtues and limitations”, Proceedings of the 7th VLDB Conference, September 1981, pp. 144-154.

[2] D. J. Taylor, “How big can an atomic action be?”, Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems, Los Angeles, January 1986, pp. 121-124.

[3] OMG, *CORBAservices: Common Object Services Specification*, Updated July 1997, OMG document formal/97-07-04.

[4] C. T. Davies, “Data processing spheres of control”, IBM Systems Journal, Vol. 17, No. 2, 1978, pp. 179-198.

[5] A. K. Elmagarmid (ed), “Transaction models for advanced database applications”, Morgan Kaufmann, 1992.

[6] H. Garcia-Molina and K. Salem, “Sagas”, Proceedings of the ACM SIGMOD International Conference on the Management of Data, 1987.

[7] S. K. Shrivastava and S. M. Wheater, "Implementing fault-tolerant distributed applications using objects and multi-coloured actions", Proc. of 10th Intl. Conf. on Distributed Computing Systems, ICDCS-10, Paris, June 1990, pp. 203-210.

[8] G.Weikum, H.J.Schek, "Concepts and Applications of Multilevel Transactions and Open Nested Transactions", in Database Transaction Models for Advanced Applications, ed. A.K. Elmagarmid, Morgan Kaufmann, 1992.

[9] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Gunthor and C. Mohan, "Advanced transaction models in workflow contexts", Proc. of 12th Intl. Conf. on Data Engineering, New Orleans, March 1996.

[10] OMG, *Additional Structuring Mechanisms for the OTS Specification*, September 2000, document orbos/2000-06-19.

[11] OMG, *Additional Structuring Mechanisms for the OTS,* RFP, May 1999, OMG document orbos/99-05-17.

[12] M. C. Little, D. McCue and S. K. Shrivastava, "Maintaining information about persistent replicated objects in a distributed system", Proc. of 13th Intl. Conf. on Distributed Computing Systems, ICDCS-13, Pittsburgh, May 1993, pp. 491-498.

[13] J. Xu, A. Romanovsky and B. Randell, "Concurrent exception handling and resolution in distributed object systems", IEEE Trans. on Parallel and Distributed Systems, Vol. 11, No. 10, 2000.

[14] B. Bennett, B. Hahm, A. Leff, T. Mikalsen, K. Rasmus, J. Rayfield and I. Rouvellou, "A distributed object oriented framework to offer transactional support for long running business processes", Middleware 2000, (J. Sventek and G. Coulson eds.), LNCS 1795, pp. 331-348, 2000.

[15] S.M. Wheater, S.K. Shrivastava and F. Ranno, "A CORBA Compliant Transactional Workflow System for Internet Applications", Middleware 98, (N. Davies, K. Raymond, J. Seitz, eds.), Springer-Verlag, London, 1998, ISBN 1-85233-088-0, pp. 3-18.

[16] OASIS BTP 1.0 specification, May 2002, http://www.oasis-open.org/committees/business-transactions/documents/specification/2002-06-03.BTP_cttee_spec_1.0.pdf

[17] "Work Service Area for J2EE", http://www.jcp.org/jsr/detail/149.jsp

[18] "Java Message Service API 1.0.2", http://java.sun.com/products/jms/

[19] "J2EE Activity Service for Extended Transactions", http://www.jcp.org/jsr/detail/95.jsp

[20] "Java Transaction API 1.0.1", http://java.sun.com/products/jta/

[21] M. Little et al, "A framework for implementing business transactions on the Web", Hewlett-Packard initial submission to OASIS BTP, http://www.oasis-open.org/committees/business-transactions/documents/HP_submission.pdf