Modules and Interfaces 2

Contents

This chapter contains the following sections.

Section Title	Page
“The Activity Service Modules”	2-1
“Activity Service Interfaces”	2-10
“Distributing Context Information”	2-38
“The User’s View”	2-40
“The Implementor’s View”	2-46

2.1 The Activity Service Modules

2.1.1 Overview

The set of CORBA services which support the Activity Service Model presented earlier are supported in the **CosActivity**, **CosActivityAdministration**, and **CosActivityCoordination** modules. This chapter shall describe the datatypes, exceptions, and interfaces provided by these different modules.

2.1.2 Datatypes

The **CosActivity** module defines the following datatypes:

2.1.2.1 GlobalId

This sequence of octets is used to uniquely identify the Activity. It is implementation dependent as to the information that may be contained within **GlobalId**.

```
typedef sequence<octet> GlobalId;
```

2.1.2.2 Status

During the existence of the activity its status will either be running, completing, or completed.

```
enum Status
{
    StatusActive,
    StatusCompleting,
    StatusCompleted,
    StatusError,
    StatusNoActivity,
    StatusUnknown
};
```

The meaning of each of the above values is given below:

- **StatusActive**: An Activity is associated with the target object and the Activity is in the active state. An implementation returns this status after an Activity has been started and prior to its beginning completion.
- **StatusCompleting**: An Activity is associated with the target object and it is in the process of completing. An implementation returns this status if it has started to complete, but has not yet finished the process. This value indicates that the activity may be performing activity specific work required to determine its final completion status. An activity must enter this state prior to completion, even if this state does nothing.
- **StatusCompleted**: An Activity is associated with the target object and it has completed. The actual outcome of the completed Activity will depend upon the type of Activity (e.g., a transactional Activity may complete in a *Committed*, or *RolledBack* state). Obtaining such states will be application specific.
- **StatusError**: An Activity is associated with the target object but it is unable to proceed as one or more of its entities are not available. The Activity may be in an inconsistent state.
- **StatusNoActivity**: No Activity is currently associated with the target object. This will occur after an Activity has completed, or before the first Activity is created.
- **StatusUnknown**: An Activity is associated with the target object, but the Activity Service cannot determine its current status. This is a transient condition, and a subsequent invocation will ultimately return a different status.

Figure 2-1 indicates the transitions that an Activity can undergo.

Figure 2-1 Activity UML state diagram

2.1.2.3 *CompletionStatus*

```
enum CompletionStatus
{
    CompletionStatusSuccess,
    CompletionStatusFail,
    CompletionStatusFailOnly
};
```

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.

2.1.3 *Structures*

2.1.3.1 *ActivityInformation*

```
struct ActivityInformation
{
  GlobalId activityId;
  CompletionStatus status;
    Outcome final_outcome;
```

```
};
```

The **ActivityInformation** structure is encoded within the **application_specific_data** field of the Signals sent by the ChildLifetime and Synchronization SignalSets.

2.1.3.2 Signal

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

An Activity may enable Signal objects to be transmitted to entities to inform them about activity specific events. Activity specific information (e.g., about how the Activity terminated) is encoded within the Signal.

signal_name is an identifier for the Signal, and can be used to determine the meaning of the Signal. It is invalid for this field to be nil. This name must be unique within the context of the **SignalSet**.

signal_set_name is the name of the SignalSet this Signal is associated with. It is invalid for this field to be nil. These names must be unique, and adhere to the following naming convention: <domain>.<company>.<…>; so, for example, “com.ibm.fred.otssignals”.

The **application_specific_data** field may be used to encode additional application specific information.

Predefined signal types include:

- *preCompletion*: the recipient is informed that the Activity is about to complete. This Signal will only be called if the Activity’s completion status is **CompletionStatusSuccess**. The Activity’s completion status and its identity is encoded within the Signal via the **ActivityInformation** structure. The **ActivityInformation final_outcome** is nil for this Signal.
- *postCompletion*: the recipient is informed that the Activity has completed. Information about the Activity’s completion status, which may have changed since preCompletion, is encoded within the Signal. The Activity’s completion status, final **Outcome**, and its identity is encoded within the Signal via the **ActivityInformation** structure.
- *childBegin*: the recipient is informed that the Activity has begun. The Activity’s completion status and its identity is encoded within the Signal via the **ActivityInformation** structure.The **ActivityInformation final_outcome** is nil for this Signal.

An Activity Service implementation will not modify the **application_specific_data** field of any Signal.

2.1.3.3 Outcome

```
struct Outcome
{
   string outcome_name;
   any    application_specific_data;
};
```

When an Action receives a specific Signal it returns an Outcome that represents the result of its having dealt with the Signal. When an Activity completes, an Outcome may be returned to the application in order for it to determine the final status of the Activity.

outcome_name is an identifier for the Outcome, and can be used to determine the meaning of the Outcome. It is invalid for this field to be nil.

The **application_specific_data** field may be used to encode additional application specific information.

Actions are required to use the ActionError exception to indicate that some failure occurred during Signal processing. This exception is mapped onto the pre-defined Outcome "ActionError." Other system exceptions (such as the failure of an Action to respond to a given Signal), are mapped onto the pre-defined Outcome "ActionSystemException," and information about the exception is encoded within the **application_specific_data** field.

2.1.3.4 ActivityIdentity and ActivityContext

```
struct ActivityIdentity
{
   unsigned long type;
   long timeout;
   ActivityCoordinator coord;
   sequence <octet> ctxId;
   sequence <PropertyGroupIdentity> pgCtx;
   any activity_specific_data;
};

struct ActivityContext
{
   sequence <ActivityIdentity> hierarchy;
   any invocation_specific_data;
};
```

Activities may be composed of other Activities. If an activity is started within the scope of an already running Activity, then it will automatically be nested within that Activity (i.e., it will be a child Activity). Thus, the execution of a series of Activities may form a hierarchy. When entities within an Activity invoke objects in other address spaces, information about the context in which these invocations are made must flow with the invocation.

Each activity may have an arbitrary number of transactions running within it (or none), and the top entities within such a hierarchy may be transactions. A receiving execution domain may be required to recreate the imported activity context such that recreated activities are running within the right (recreated) transaction scopes. Transaction context propagation issues are dealt with by the Object Transaction Service specification and will not be discussed here. However, sufficient information needs to be shipped by the exporting Activity Service to enable importing environments to recreate the sent Activity context, such that recreated Activities and transactions are nested in the importing environment in the same way they are in the exporting environment.

If an activity context is sent on an outward request, a context may be returned on the response. This returned context need not be the same as was originally sent, e.g., low-cost interposition information may be encoded within the context and piggybacked on the response. For a remote request that completes without exception, the absence of an Activity service context on a response should be taken to mean that the context has not been changed by the target domain. This should be true even in the case where a transaction context is present on both request and response.

The objects using a particular Activity Service implementation in a system form an *Activity Service domain*. Within the domain, the structure and meaning of the activity context information can be private to the implementation. When leaving the domain, this information must be translated to a common form if it is to be understood by the target Activity Service domain. Therefore, an Activity context (hierarchy) is represented by the **ActivityContext**, which is an ordered sequence of **ActivityIdentitys**. The first element in the sequence represents the current Activity/transaction, and the last represents the root of the hierarchy.

The *type* field, which must be a positive, non-zero value, is used to indicate the type of the element for which the information is being maintained. Currently supported values are:

- 1: the element in the hierarchy is a transaction.
- 2: the element in the hierarchy is an Activity.

An element within the hierarchy is uniquely identified by an instance of **ActivityIdentity**. If the *type* field indicates that the element is an Activity, then the *coord* field will be set, and *ctxId* will be the Activity's unique identifier. If the type field indicates that the element is a transaction, then the *coord* field will be nil, and the *ctxId* will be the tid portion of the **CosTransactions::otid_t** representation for the OTS transaction at this level in the hierarchy.

Although the **ActivityIdentity** contains a field for the tid portion of the transaction's **CosTransactions::otid_t**, this is merely so that the position of any transaction context can be recorded relative to the Activity context (if any) within which it was started. Each nested transaction is represented by exactly one **ActivityIdentity**, which marks the sub-transaction's position within the hierarchy.

In order to reduce the amount of context information which is transmitted between execution domains where nested transactions are used, the **ActivityContext** structure need only contain information on an activity's most deeply nested transaction, since this is sufficient to be able to recreate the entire activity/transaction hierarchy.

The Activity Service uses the **PropertyGroupManagers** to fill in the *pgCtx* field.

The timeout field indicates the application specific timeout associated with the activity or transaction when it was created. (If this instance represents a subtransaction, then this field will be -1.) If the activity or transaction has not completed within this time period, then it will be completed with **CompletionStatusFail**.

Additional information may be encoded within the **activity_specific_data** and **invocation_specific_data** fields. It is legal for these fields to contain an empty any. An implementation must not rely on the data that was sent with an outbound context being available on the reply context. The **invocation_specific_data** is meant to carry information which is required for a specific implementation of the service. Because this information is specific to a given implementation of the Activity Service it is illegal for an importing domain that is different from the exporting domain to use this field. To ensure integrity of the application (specifically in the case of loop-backs between foreign and native domains), a domain which does not understand the **invocation_specific_data** within an activity context must replace it with an empty any. Such a domain is free, however, to replace the data with data specific to itself. The **activity_specific_data** is meant to carry information which is required for an implementation of a specific extended transaction model. If an importing domain implements a different extended transaction model than the exporting domain, i.e., it does not understand the **activity_specific_data**, then it must not use the context, and should throw BAD_CONTEXT.

Type values for Activities supporting specific extended transaction models will be defined in the future. Each specific type will also define the format of the **activity_specific_data** that may be propagated as part of the ActivityIdentity structure in the service context.

2.1.3.5 *PropertyGroupIdentity*

```
struct PropertyGroupIdentity
{
   string property_group_name;
   any context_data;
};
```

PropertyGroups form part of the Activity Service context. It is dependent upon the implementations of each **PropertyGroup** how information about them flows in the context. Therefore, it is up to the **PropertyGroupManager** to marshal and unmarshal **PropertyGroups** appropriately. The **PropertyGroupIdentity** structure is used to encapsulate this marshaled form of the **PropertyGroup**.

property_group_name is the name of the **PropertyGroup**. Implementations must ensure that such names are unique within the required domain.

context_data represents the marshaled form of the **PropertyGroup**.

2.1.4 Exceptions

The **CosActivity** and **CosActivityAdministration** modules define the following exceptions that can be raised by an operation.

NoActivity Exception

The NoActivity exception is raised by methods on the **Current** interface where an Activity is required to be active on the thread but none is.

ActivityPending Exception

The ActivityPending exception is raised if an attempt is made to complete the Activity when it is active on a thread other than the calling thread.

ActivityNotProcessed

The ActivityNotProcessed exception is raised to indicate that it was not possible to complete the processing of signals from a *completion* or *broadcast* SignalSet.

InvalidToken Exception

The InvalidToken exception is raised by **Current::resume** if the specified **ActivityContext** is not valid or is nil.

AttributeAlreadyExists Exception

The AttributeAlreadyExists exception is raised by **PropertyGroupAttributes::set_attribute** if the specified attribute is already set.

NoSuchAttribute Exception

The NoSuchAttribute exception is raised by **PropertyGroupAttributes::get_attribute** if the specified attribute does not exist.

ActionError Exception

The ActionError exception is raised by the Action during signal processing if it encounters an error it cannot handle.

AlreadyDestroyed Exception

The AlreadyDestroyed exception is raised by an interface if there are multiple attempts to destroy it.

ActionNotFound Exception

The ActionNotFound exception is raised by the **ActivityCoordinator** if an attempt is made to remove an Action it has no information about.

SignalSetUnknown Exception

The SignalSetUnknown exception is raised by the **ActivityCoordinator** if it is instructed to use a specified **SignalSet** it does not know about.

SignalSetAlreadyRegistered Exception

The SignalSetAlreadyRegistered exception is raised by the **ActivityCoordinator** if multiple attempts to register a **SignalSet** are made.

SignalSetActive Exception

The SignalSetActive exception is raised by the **SignalSet** when an attempt is made to obtain its final status before the **SignalSet** has completed producing Signals.

SignalSetInactive Exception

The SignalSetInactive exception is raised by the **SignalSet** if an attempt is made to use the **SignalSet** without having first called **get_signal** or **set_signal**.

PropertyGroupUnknown Exception

The PropertyGroupUnknown exception is raised if an attempt it made to obtain an unknown **PropertyGroup**.

PropertyGroupAlreadyRegistered Exception

The PropertyGroupAlreadyRegistered exception is raised if multiple attempts to register a **PropertyGroup** are made.

PropertyGroupNotRegistered Exception

The PropertyGroupNotRegistered exception is raised if an attempt is made to unregister a **PropertyGroup** that has not previously been registered.

ChildContextPending Exception

The ChildContextPending exception is raised if an attempt is made to successfully complete an Activity when it still has active child Activities.

InvalidState Exception

The InvalidState exception is raised to indicate that the completion status of the Activity is incompatible with the attempted invocation.

InvalidParentContext Exception

The InvalidParentContext exception is raised either if an attempt is made to resume a suspended context within a different hierarchy than that which it was originally suspended from, or an attempt is made to call **CosActivity::suspend** on an Activity that is nested within a transaction.

TimeoutOutOfRange Exception

The TimeoutOutOfRange exception is raised if an attempt it made to associated an invalid timeout with a newly created Activity.

InvalidContext Exception

The InvalidContext exception is raised to indicate that a context could not be correctly imported.

INVALID_ACTIVITY Exception

The INVALID_ACTIVITY system exception may be raised on the Activity or Transaction services' resume methods if a transaction or Activity is resumed in a context different to that from which it was suspended. It is also raised when an attempted invocation is made that is incompatible with the Activity's current state.

ACTIVITY_COMPLETED Exception

The ACTIVITY_COMPLETED system exception may be raised on any method for which Activity context is accessed. It indicates that the Activity context in which the method call was made has been completed due to a timeout of either the Activity itself or a transaction that encompasses the Activity, or that the Activity completed in a manner other than that originally requested.

ACTIVITY_REQUIRED Exception

The ACTIVITY_REQUIRED system exception may be raised on any method for which an Activity context is required. It indicates that an Activity context was necessary to perform the invoked operation, but one was not found associated with the calling thread.

2.2 Activity Service Interfaces

2.2.1 SignalSet Interface

```
interface SignalSet
{
   readonly attribute string signal_set_name;

   Signal get_signal(inout boolean lastSignal);

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

   Outcome get_outcome () raises(SignalSetActive);

   void set_completion_status (in CompletionStatus cs);
```

```
    CompletionStatus get_completion_status ();

      void set_activity_coordinator (in ActivityCoordinator coord)
                                    raises(SignalSetActive);

    void destroy() raises(AlreadyDestroyed);
};
```

The **SignalSet** is used to define the individual signals that are broadcast to the Action objects. Actions that have been registered as being interested in a specific **SignalSet** are sent Signals from that **SignalSet**. Typically once all Actions have received a given Signal, the **SignalSet** is asked for the next Signal to be sent to all of the Actions, if any.

If a **SignalSet** fails to produce Signals (e.g., it is physically remote from the **ActivityCoordinator** and fails to respond to invocations), then the completion status of the Activity is set to CompletionStatusFailOnly, and the **ActivityCoordinator** should act accordingly.

If a **SignalSet** fails to produce Signals (e.g., it is physically remote from the **ActivityCoordinator** and fails to respond to invocations), then the pre-defined *org.omg.CosActivity.Failure* **SignalSet** should be used instead. All pre-defined **SignalSet** are restricted to being located in the same domain as the **ActivityCoordinator** using them. Any Actions registered with an interest in the unreachable **SignalSet** will be sent Signals produced from the Failure **SignalSet**.

Once the Activity has begun to complete (the **ActivityCoordinator** has retrieved the first Signal from a **SignalSet**), the status of the Activity is under the control of the SignalSets, and cannot be changed directly by any other entity.

Signals are specified as members of **SignalSet**s. As mentioned previously, it is envisioned that the majority of Signals and **SignalSet**s will be defined by the higher-level extended transaction systems that use this Activity framework. Only such systems have the necessary application and activity specific knowledge to impose structure on the meaning of specific Signals and **SignalSet**s. However, there are a small set of pre-defined signal sets and their associated signals, which are provided by implementations of the Activity Service:

- *org.omg.CosActivity.ChildLifetime*: childBegin
- *org.omg.CosActivity.Synchronization*: preCompletion, postCompletion
- *org.omg.CosActivity.Failure*: initialFailure, finalFailure

These pre-defined **SignalSet**s are implicitly associated with every Activity when it is created, and an application need not register them itself (i.e., no call to **ActivityCoordinator::add_signal_set** is required).

org.omg.CosActivity.ChildLifetime

The ChildLifetime **SignalSet** is invoked by the parent when a sub-Activity is begun. There are no pre-defined Outcomes introduced by this **SignalSet**. If an Action error occurs during childBegin (e.g., the ActionError exception is thrown), then the child's

Activity completion status will be set to **CompletionStatusFailOnly**; it is up to the parent activity (or the application) to determine whether such a failure should cause the parent activity's completion status to be changed.

ChildLifetime signals are distributed from the environment in which the child activity is started. If the parent of a sub-Activity is not a root Activity (i.e., it is an interposed subordinate), then any Actions registered with the (upstream) superior **ActivityCoordinator** do not receive these signals and are unaware of the child activity.

If an indication of the termination of an activity is required, then the *org.omg.CosActivity.Synchronization* **SignalSet** should be used on the respective activity.

The child activity is active on the thread when childBegin is issued.

org.omg.CosActivity.Synchronization

The Synchronization **SignalSet** has a similar role to that of Synchronization objects within the OTS (i.e., it is invoked before and after completion of the Activity). Likewise, the completion status of an Activity may be changed by the Actions registered with this **SignalSet**, such that the Activity's outcome when postCompletion is called may be different to that when preCompletion was invoked. If an Action error occurs during preCompletion (e.g., the ActionError exception is thrown), then the Activity completion status will be set to **CompletionStatusFailOnly**. There is no effect on the completed Activity if a failure occurs during postCompletion.

The preCompletion **SignalSet** is only sent if the Activity's completion status is **CompletionStatusSuccess**. In the event of no crash failures that prevent the **ActivityCoordinator** from completing its work, postCompletion is sent regardless of the Activity's completion status.

If there are any Actions registered with it, then the Synchronization **SignalSet** will be called prior to using any application specific **SignalSet**. The pre-defined Outcomes "preCompletionSuccess" and "preCompletionFailed" may be produced by an Action in response to the preCompletion signal. If an Action fails to respond to preCompletion or a failure occurs, or the Synchronization **SignalSet** receives the preCompletionFailed Outcome from an Action and the completion status of the Activity is changed to **CompletionStatusFailOnly**.

The following pre-defined **Outcomes** may also be produced by this **SignalSet**, typically when the **SignalSet** is invoked in the environment of a subordinate **ActivityCoordinator**:

- "preCompletionActivityPending". This **Outcome** is returned if the Activity is concurrently active on another thread when the preCompletion signal is received.
- "preCompletionChildContextPending". This **Outcome** is returned if there is an outstanding child Activity, or a transaction context encompassed within the Activity, when the preCompletion signal is received.

These outcomes indicate that there is work outstanding that needs to be completed before the preCompletion signal can be processed. These outcomes must be processed by an **ActivityCoordinator** in such a way that the application which requested the completion of the Activity receives an ActivityPending or ChildContextPending exception.

If the **SignalSet** decides that the next Signal (postCompletion) is required or normal processing of preCompletion has finished, then the implementation of the Activity Service must first invoke the application specific **SignalSet** (if any) with the (potentially new) completion status obtained from **get_completion_status** of the Synchronization **SignalSet** (i.e., postCompletion is not called immediately). When the application **SignalSet** has finished producing Signals the postCompletion Signal should be sent to the registered Actions. Errors during postCompletion have no effect on the outcome of the Activity.

The completing activity is active on the thread when preCompletion is sent. However, it is inactive on the thread when postCompletion is generated by the **SignalSet**.

org.omg.CosActivity.Failure

The Failure **SignalSet** is used by the **ActivityCoordinator** if an application **SignalSet** cannot be reached during signaling. The Failure **SignalSet** produces two signals - *initialFailure* and *finalFailure*.

initialFailure indicates that the application **SignalSet** could not be contacted but that the problem may be transient. An Action that receives the *initialFailure* **Signal** should respond with one of two pre-defined Outcomes “Failed” or “FailureRetry”. Any Action that responds with Failed will not receive any further **Signals**. Any Action that responds with FailureRetry is indicating that it wishes the **ActivityCoordinator** to continue to retry contacting the application **SignalSet**. If contact is subsequently made, signaling with the application **SignalSet** may continue.

An Activity service implementation may chose at which point, if any, to abandon its attempt to contact the application **SignalSet**. At this point the Failure **SignalSet** is asked to produce the *finalFailure* **Signal** which is distributed to any remaining Actions for them to perform whatever processing is appropriate to them in this situation. The Failure **SignalSet** ignores any Outcome returned in response to this **Signal**. The Activity service changes the Activity status to **StatusUnknown** prior to distributing the *initialFailure* signal. The Activity service changes the Activity status to **StatusError** prior to distributing the *initialFailure* signal. If the application **SignalSet** does not complete its signaling, the **ActivityCoordinator** raises the *org.omg.CosActivity.ActvityNotProcessed* exception on the **complete_activity** or **process_signal_set** method that triggered the signaling and this exception is returned to the application through the *Current* **complete**, **complete_with_status** or **broadcast** methods.

Both *initialFailure* and *finalFailure* **Signals** have the name of the failed **SignalSet** as their **signal_set_name** field, inorder that recipients can determine which **SignalSet** the failure corresponds to.

set_completion_status

This method is used to provide the Activity's completion status to the **SignalSet** during its generation of Signals, such that it can use the status to determine whether or not the Activity is completing when it produces **Signals**.

get_completion_status

This returns the Activity's completion status as the **SignalSet** has recorded it (and as it may have been modified during Signal processing). If the **SignalSet** has not generated any Signals (i.e., is inactive), then SignalSetInactive is thrown.

signal_set_name

Returns the name of this **SignalSet**. These names must be unique, and adhere to the following naming convention: <domain>.<company>.<module>.<…>; so, for example, "com.ibm.fred.otssignals".

get_ signal

Returns the Signal to be sent to the Action objects registered for this signal set. The Signal returned may depend upon the responses received from Actions that have been sent previous signals. If nil is returned, or the boolean output parameter *lastSignal* is true, then this indicates that no other signals are to be sent and the **SignalSet** will not be asked for further Signals. It is therefore valid for a **SignalSet** to indicate no further Signals are available either through *lastSignal* or returning nil. Whenever either of these conditions is encountered, the coordinator must not call the **SignalSet** again.

set_response

This method is called to notify the **SignalSet** of the response (the Outcome) from the Action object. It is valid for the Outcome parameter to be nil. The SignalSet returns a boolean to indicate whether or not the Action that returned the response should be informed of any further signals from this signal set; if the value is *true* then the Action continues to receive Signals for this **SignalSet**, otherwise the Action is disassociated from the **SignalSet**, i.e., this is equivalent to it being removed. If **nextSignal** is true then no further work with the current Signal should be performed and the registered Actions should be sent the next Signal belonging to this SignalSet. For example, if an Action returns a failure condition on some Signal (say "prepare"), which indicates that it is pointless to send further signals of this type to other Actions, **nextSignal** would be set to true. The next signal obtained from **get_signal** may then be different from that which would have been obtained if no failure condition had been observed. If **get_signal** has not yet been called, then SignalSetInactive will be thrown.

get_outcome

Returns the final outcome of the **SignalSet**; it is valid for this value to be nil. If the **SignalSet** has start producing **Signals** but not finished producing then, then the SignalSetActive exception will be thrown.

set_activity_coordinator

This method is used by the **ActivityCoordinator** to pass a reference to itself to the **SignalSet**. The **SignalSet** can then use this to obtain references to all registered Actions in order to satisfy persistence requirements, for example, and optimisations such as one-phase commit. For example, consider the case of a two-phase commit **SignalSet**: once prepare **Signals** have been sent and acknowledged successfully by **Actions**, the service needs to make those **Action** references persistent (c.f. the transaction service intentions list). If the **SignalSet** has already been asked for its first Signal, then the SignalSetActive exception will be thrown, and the coordinator reference will be ignored.

destroy

This method is invoked when the **SignalSet** is no longer required by the Activity service. If the **SignalSet** has already been destroyed, or is being destroyed, then the AlreadyDestroyed exception will be thrown. Any exception thrown will not affect the outcome of the activity.

2.2.2 SubordinateSignalSet Interface

```
interface SubordinateSignalSet : SignalSet
{
    void set_signal (in Signal sig);
    Outcome get_current_outcome () raises(SignalSetInactive);
};
```

A domain that contains an interposed subordinate ActivityCoordinator can support Actions registering at that subordinate ActivityCoordinator with an interest in, say, SignalSet “X”. The subordinate ActivityCoordinator must use a specialised implementation of X that supports a SubordinateSignalSet interface.

set_signal

Sets the Signal to be sent to the Action objects registered for this SubordinateSignalSet. This method is called by a subordinate ActivityCoordinator when it receives a Signal from its superior. The subordinate ActivityCoordinator distributes this Signal to each appropriate Action and passes each Action Outcome back to the SubodinateSignalSet via the set_response method. The SubordinateSignalSet produces a combined Outcome for the set Signal and this is returned by the subordinate ActivityCoordinator to its superior. Any system exceptions raised by the SubordinateSignalSet should be converted to an ActionError by the subordinate ActivityCoordinator.

get_current_outcome

Returns an intermediate outcome of the SubordinateSignalSet. This may be called after the processing of each Signal and is used by a subordinate ActivityCoordinator to obtain an Outcome to return to its superior in response to a received Signal. If the SignalSet has not been initialized, for example by a call to set_signal, then the SignalSetInactive exception will be thrown.

2.2.3 Action Interface

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

   void destroy() raises(AlreadyDestroyed);
};
```

Instances of the **Action** interface may be registered with running Activities, such that when the Activities require Signal processing, the registered Actions will be invoked. When an Action is invoked, it is passed a Signal object that can be used to do application specific work.

An Action may receive many different Signals from different SignalSets.

process_signal

This method is invoked by the Activity service during signal processing. The Action returns an Outcome to indicate the outcome of the processing operation.

destroy

This method is invoked when the Action is no longer required by the Activity service, e.g., because the Activity it is registered with has completed. This method is only called on Actions that did not register with the *org.omg.CosActivity.Synchronization* SignalSet. An Action may determine that it is no longer required by the activity is has been registered with before *destroy* is called. It is therefore legal for an Action to remove itself before this method has been invoked by the activity. As a result, the service implementation will ignore OBJECT_NOT_EXIST. It is implementation dependant as to the result of receiving other system exceptions, but they can have no affect on the completed activity.

2.2.4 ActivityToken Interface

```
interface ActivityToken
{
   ActivityContext get_context ();
   void destroy() raises(AlreadyDestroyed);
};
```

In order to allow for efficient implementations of inter- and intra- process Activity coordination and control, the Activity Service provides two different representations for the **ActivityContext**. When an Activity is suspended from an active thread, an **ActivityToken** is returned which is a handle to the activity context and *is only valid within the obtaining execution domain*. This can later be used to resume the Activity on the same, or other thread. The **ActivityToken** is implicitly associated with a single Activity, and thus the context it represents can be obtained from it. This is preferable to having to deal with the entire **ActivityContext** when suspending and resuming in a local environment.

get_context

Returns the **ActivityContext** represented by this **ActivityToken**. If the token was obtained by a call to **CosActivity::suspend_all**, then the entire hierarchy context will be returned, otherwise only the current context.

destroy

This method is invoked when the **ActivityToken** is no longer required by the Activity service. If the **ActivityToken** has already been destroyed, or is being destroyed, the AlreadyDestroyed exception will be thrown. Any exception thrown will have no affect on the activity's outcome.

2.2.5 *ActivityCoordinator Interface*

```
interface ActivityCoordinator
{
   Outcome complete_activity(in string signal_set_name,
                             in CompletionStatus cs)
        raises(ActivityPending, ChildContextPending,
               SignalSetUnknown, ActivityNotProcessed);
   Outcome process_signal_set(in string signal_set_name,
                              in CompletionStatus cs)
         raises(SignalSetUnknown, ActivityNotProcessed);

   void add_signal_set (in SignalSet signal_set)
         raises(SignalSetAlreadyRegistered);
   void remove_signal_set (in string signal_set_name)
         raises(SignalSetUnknown);

   void add_action(in Action act, in string signal_set_name,
              in long priority) raises(SignalSetUnknown);
   void remove_action(in Action act, in string signal_set_name)
                              raises(ActionNotFound);

   void add_actions(in ActionSeq acts, in string signal_set_name,
              in long priority) raises(SignalSetUnknown);
   ActionSeq remove_actions(in ActionSeq acts, in string signal_set_name);

   void add_global_action(in Action act, in long priority);
   void remove_global_action(in Action act) raises(ActionNotFound);

   long get_number_registered_actions(in string signal_set_name)
                              raises(SignalSetUnknown);
   ActionSeq get_actions(in string signal_set_name)
                 raises(SignalSetUnknown);

   ActivityCoordinator get_parent_coordinator ();

   GlobalId get_global_id ();
```

```
    Status get_status ();
    Status get_parent_status ();
    string get_activity_name ();

    boolean is_same_activity (in ActivityCoordinator ac);

    unsigned long hash_activity ();

    void destroy() raises(AlreadyDestroyed);
};
```

The **ActivityCoordinator** is responsible for coordinating the interactions between Activities through Signals, SignalSets, and Actions (i.e., in the model presented earlier it "ties" up the Actions of Activities).

It is not strictly necessary for an implementation of the Activity Service to create an **ActivityCoordinator** prior to distributing a context between execution environments in which it was begun. Implementations of the Activity Service may restrict the use of the **ActivityCoordinator** in certain environments, such as a light-weight client.

Each Activity may be managed by at most one **ActivityCoordinator**.

Implementations of the Activity Service may use interposition to reduce the number of network messages required to complete an activity.

Once the **ActivityCoordinator** has used all of the Signals generated by the **SignalSet**, it may invoke the destroy operation on all registered Actions, including those that may have been registered with other **SignalSet**s and hence not received Signals during Activity termination.

complete_activity

This instructs the **ActivityCoordinator** to complete the Activity using the specified **SignalSet** when sending signals to registered Actions, with the provided completion status. If the **SignalSet** is unknown, the SignalSetUnknown exception will be raised; it is valid for the specified **SignalSet** to be null.

If an Action throws the ActionError or System exception, then it is dependent upon the **SignalSet** implementation as to whether the **ActivityCoordinator** stops sending signals to other registered Actions; this may depend upon the type of Signal that was being processed at the time the exception occurred.

If the Action throws ActionError or any system exception, then this may be mapped into either the pre-defined Outcomes "ActionError" or "ActionSystemException" respectively and passed to the **SignalSet**; for system exceptions, the exception is also passed in the **application_specific_data** portion of the Outcome.

If the **ActivityCoordinator** is currently processing Signals when **complete_activity** is invoked, or has already completed, the INVALID_ACTIVITY exception is thrown. Successful completion of this method causes the Outcome, if any, of the **SignalSet** processing to be returned. It is valid for this return value to be nil. It is invalid to attempt to explicitly use the Synchronization or ChildLifetime **SignalSet**s, and

BAD_OPERATION will be thrown under these circumstances. The pre-defined **SignalSet**s Synchronization and ChildLifetime will be automatically invoked during Activity completion if Actions have registered in them.

If there are any encompassed active or suspended Activities or transactions, and the completion status is **CompletionStatusSuccess**, then ChildContextPending is raised. If the completion status is **CompletionStatusFail** or **CompletionStatusFailOnly**, any encompassed active or suspended Activities will have their completion status set to **CompletionStatusFailOnly** and transactions will be marked as **rollback_only**.

If the thread from which the **complete_activity** call is made is not the only thread on which the Activity is active, then the ActivityPending exception is raised. It is recommended that this operation not be called directly.

The ActivityNotProcessed exception is raised in the event that the signals required to complete this operation could not be produced.

process_signal_set

This instructs the **ActivityCoordinator** to use the specified **SignalSet** when sending signals to registered Actions, with the provided completion status; this method cannot be used to complete the Activity, and **complete_activity** should be used instead. If the **SignalSet** is unknown the SignalSetUnknown exception will be raised; it is valid for the specified **SignalSet** to be null.

If an Action throws the ActionError or a System Exception, then it is dependent upon the **SignalSet** implementation as to whether the **ActivityCoordinator** stops sending signals to other registered Actions; this may depend upon the type of Signal that was being processed at the time the exception occurred.

If the Action throws ActionError or any system exception, then this may be mapped into either the pre-defined Outcomes “ActionError” or “ActionSystemException” respectively and passed to the SignalSet; for system exceptions, the exception is also passed in the **application_specific_data** portion of the Outcome.

If the **ActivityCoordinator** is currently processing Signals when **process_signal_set** is invoked, or has already completed, the INVALID_ACTIVITY exception is thrown. Successful completion of this method causes the Outcome, if any, of the **SignalSet** processing to be returned. It is valid for this return value to be nil. It is invalid to attempt to explicitly use the Synchronization or ChildLifetime **SignalSet**s, and BAD_OPERATION will be thrown under these circumstances. It is recommended that this operation not be called directly.

The ActivityNotProcessed exception is raised in the event that the signals required to complete this operation could not be produced.

add_signal_set

This method registers the specified **SignalSet** with the **ActivityCoordinator**. If the **SignalSet** has already been registered then the SignalSetAlreadyRegistered exception will be raised. If the **ActivityCoordinator** is in use (i.e., is processing Signals or has completed), then the INVALID_ACTIVITY exception is thrown.

remove_signal_set

This method removes the specified **SignalSet** from the **ActivityCoordinator**. If the Activity has begun completion, has completed, or is in the process of using the specified **SignalSet**, then the INVALID_ACTIVITY exception is thrown. If the **SignalSet** is not known, then SignalSetUnknown will be raised. It is invalid to attempt to remove the pre-defined **SignalSet**s org.omg.CosActivity.Synchronization and org.omg.CosActivity.ChildLifetime, and BAD_OPERATION will be thrown.

add_action

This method registers the specified Action with the **ActivityCoordinator** and **SignalSet** such that when a Signal which is a member of the **SignalSet** is sent, the Action will be invoked with that Signal. If multiple Actions are registered, then *priority* may be used to place an order on how they will be invoked when signals are sent: higher priority Actions will occur first in the Action list, and hence be invoked before other, lower priority, Actions. The priority value must be a positive value; a value of zero means that the Activity Service implementation is free to place the Action at any point in the Action list. If the **SignalSet** is not known about, then the SignalSetUnknown exception is thrown. If the Activity has begun completion, or has completed, then the INVALID_ACTIVITY exception is thrown. If the specified Action is registered multiple times for the same SignalSet then it will be invoked multiple times with the Signals from that SignalSet.

add_actions

This method registers a number of Actions with the **ActivityCoordinator**; such Actions are assumed to be already prioritized within the sequence. If multiple Actions are registered, then *priority* may be used to place an order on how they will be invoked: higher priority numbers will be invoked before lower priority numbers. The priority value must be a positive value; a value of zero means that the Activity Service implementation is free to place the Action at any point in the Action list. If the **SignalSet** is not known about, then the SignalSetUnknown exception is thrown. If the Activity has begun completion, or has completed, then the INVALID_ACTIVITY exception is thrown. If the specified Action is registered multiple times for the same SignalSet then it will be invoked multiple times with the Signals from that SignalSet.

add_global_action

This method registers the specified Action with the **ActivityCoordinator** such that when *any* Signal is sent, the Action will be invoked with that Signal (i.e., the Action is effectively registering interest in all possible **SignalSet**s). If multiple Actions are registered, then *priority* may be used to place an order on how they will be invoked: higher priority numbers will be invoked before lower priority numbers. The priority value

must be a positive value; a value of zero means that the Activity Service implementation is free to place the Action at any point in the Action list. If the Activity has begun completion, or has completed, then the INVALID_ACTIVITY exception is thrown.

remove_action

Removes the interest relationship between the specified Action and the named SignalSet. No further Signals from the named SignalSet will be sent to the specified Action. If **signal_set_name** is specified as an empty string, then the Action will be sent no further Signals from any SignalSet. If the Action has not previously been registered with the coordinator, then the ActionNotFound exception will be thrown. If the Activity has begun completion, or has completed, then the INVALID_ACTIVITY exception is thrown.

remove_actions

Removes the interest relationship between the specified Actions and the named SignalSet. No further Signals from the named SignalSet will be sent to the specified Actions. If signal_set_name is specified as an empty string, then the Actions will be sent no further Signals from any SignalSet. If any of the Actions have not previously been registered with the coordinator, then it will return references to them after removing all other Actions in the sequence. Otherwise nil will be returned. If the Activity has begun completion, or has completed, then the INVALID_ACTIVITY exception is thrown.

remove_global_action

This method removes the specified Action from the **ActivityCoordinator**. If the Action has not previously been registered with the coordinator, then it will throw the ActionNotFound exception. If the Activity has begun completion, or has completed, then the INVALID_ACTIVITY exception is thrown.

get_number_registered_actions

Returns the number of Actions that have been registered with the specified **SignalSet**.

get_actions

Returns all the Actions that have been registered with the specified **SignalSet**.

get_parent_coordinator

Returns a reference to the **ActivityCoordinator**'s parent, or null if this coordinator has no parent (i.e., is at the root of the Activity hierarchy).

get_global_id

Returns the **GlobalId** for the Activity.

get_status

Returns the current status of the associated Activity.

get_parent_status

Either returns the status of the target objects' parent Activity, or the target object's status if it is top-level (i.e., has no parent).

get_activity_name

This operation returns a printable string describing the activity. This value should only be used for debugging or tracing purposes.

hash_activity

Returns a hash code for the activity associated with the target object. Each **ActivityCoordinator** has a single hash code. Hash codes for Activities should be uniformly distributed.

is_same_activity

Returns true if, and only if, the target object and the parameter object both refer to the same activity.

destroy

This method is invoked when the **ActivityCoordinator** is no longer required by the Activity service. If the **ActivityCoordinator** has already been destroyed, or is being destroyed, then the AlreadyDestroyed exception will be thrown. Any exception thrown by *destroy* will not affect the outcome of the activity.

2.2.6 PropertyGroup

```
interface PropertyGroup
{
   readonly attribute property_group_name;

   void completed();
   void suspended();
   void resumed();

   void destroy() raises(AlreadyDestroyed);
};
```

The **PropertyGroup** interface has the same consideration as the general Activity Service interfaces, in that it attempts to be a framework from which concrete implementations can be derived. Typically a **PropertyGroup** implementation will be a mechanism for an application to distribute context information that can affect the execution of that application in the distributed environment. The distributed environment throughout which the application executes needs to have an implementation of the required **PropertyGroup** in order for the application properties to be accessed. This is a requirement that must be resolved at application deployment time, and is outside the scope of this specification.

If the Activity Service has several **PropertyGroupManager**s registered with it, then a **PropertyGroup** will be created for each one when an Activity is begun. The **PropertyGroups** need to be informed when the Activity completes so they can perform any necessary clean-up before the Activity Service deletes them.

They may, for example, pass objects by reference rather than by value and so may need to clean up those objects. If an Activity is suspended while a client has a reference to one or more of its **PropertyGroups**, then these **PropertyGroups** should be informed that they no longer represent the currently active Activity. The behavior of the **PropertyGroup** implementation under these circumstances has to be defined by the **PropertyGroup** implementation.

The implementations of **PropertyGroups** may restrict the ability for the properties to be transmitted to or used in other execution environments; at a minimum, it can be used within the creating thread.

A **PropertyGroup** represents properties as a tuple-space of attribute-value pairs.

property_group_name

This is the name of the **PropertyGroup**.

completed

This method is called by the Activity as part of its completion process to give the **PropertyGroup** the opportunity to perform any necessary clean-up work. The Activity with which this **PropertyGroup** is associated is not active on the thread when this call is made. Any parent Activity will then become active.

suspended

This method is called to inform the **PropertyGroup** that the Activity it represents has been suspended. The Activity with which this **PropertyGroup** is associated is still active on the thread when this call is made, but will be removed immediately after all *suspended* methods of registered **PropertyGroups** have been called. Any parent Activity will then become active.

resumed

This method is called to inform the **PropertyGroup** that the Activity it represents has been resumed. The Activity with which this **PropertyGroup** is associated is already resumed on the thread when this call is made.

destroy

This method is invoked when the **PropertyGroup** is no longer required by the Activity service. If the **PropertyGroup** has already been destroyed, or is being destroyed, then the AlreadyDestroyed exception will be thrown. Exceptions thrown by *destroy* have no affect on the outcome of an activity.

2.2.7 PropertyGroupAttributes

```
interface PropertyGroupAttributes
{
   string get_attribute (in string name) raises(NoSuchAttribute);
   void set_attribute (in string name, in string value)
                  raises(AttibuteAlreadyExists);
   void replace_attribute (in string name, in string value);
};
```

An instance of the **PropertyGroupAttributes** is passed as a parameter to the **register_property_group** method of **CosActivityAdministration::Current** to set/query the behavior of the registered **PropertyGroup** for the duration of its registration.

Pre-defined attribute names and their associated values include:

- **cacheable**: on input, if set to true, then this informs the Activity Service of the intention of the **PropertyGroup** implementation to cache objects in downstream servers.
- **max_send_size** and **max_receive_size**: on output this defines the maximum size of the context data the Activity Service will send or receive on behalf of the **PropertyGroup**. The **PropertyGroupManager** is not required to use this information.
- **marshal_response_update**: indicates whether or not the PropertyGroupManager should be called when an outbound response is marshalled. A value of *true* indicates that the context for the managed PropertyGroup should be updated on a response. A value of *false* indicates that the context for the managed PropertyGroup is not updated on a response so the PropertyGroupManager is not called. The default value is *false*. It may be preferable from either a security or a performance point of view not to transmit server context back to a client with a response.
- **unmarshal_response_update**: indicates whether or not the PropertyGroupManager should be called when an inbound response is unmarshalled. A value of *true* indicates that the context for the managed PropertyGroup should be updated by the response. A value of *false* indicates that the context for the managed PropertyGroup is not updated by the response so the PropertyGroupManager is not called. The default value is *false*. It may be preferable from either a security or a performance point of view not to allow the local context to be updated by changed made in a downstream node.

Note, an implementation of **PropertyGroupAttributes** may use an implementation of the OMG's Property Service specification.

get_attribute

If the specified attribute exists, then its value is returned. This value may be nil. If the attribute does not exist then the NoSuchAttribute exception is thrown.

set_attribute

If the specified attribute does not exist, then it is created with the specified value, which may be nil. Otherwise the AttributeAlreadyExists exception is thrown.

replace_attribute

If the specified attribute does not exist, then it is created with the specified value, which may be nil. If the attribute already exists, its current value is set to that provided.

2.2.8 PropertyGroupManager

```
interface PropertyGroupManager
{
   PropertyGroup create(in CosActivity::PropertyGroup parent,
                        in CosActivity::GlobalId gid);

   PropertyGroupIdentity marshal_request(in CosActivity::PropertyGroup pg);
   PropertyGroupIdentity marshal_response(in CosActivity::PropertyGroup pg);

   PropertyGroup unmarshal_request(in CosActivity::PropertyGroupIdentity mpg,
                                   in CosActivity::PropertyGroup pg,
                                   in CosActivity::PropertyGroup parent,
                                   in CosActivity::GlobalId gid);
   void unmarshal_response(in CosActivity::PropertyGroupIdentity mpg,
                           in CosActivity::PropertyGroup pg);

   void destroy() raises(CosActivity::AlreadyDestroyed);
};
```

A **PropertyGroup** implementation registers a named **PropertyGroupManager** with the Activity Service. The registered manager understands how to create a specialized instance of the **PropertyGroup** and how to marshal/unmarshal its context, which is propagated as part of the Activity service context. A **PropertyGroupManager** must be registered with the Activity service in each domain for each type of **PropertyGroup** that is accessed via the **get_property_group** method of the *Current* interface.

create

Returns a reference to a new instance of the **PropertyGroup** specialization. This method is called by the Activity Service when a new Activity is started. A parent of nil indicates that this is the top most Activity. The gid is that of the **ActivityGroup** that is being begun. It is implementation dependent as to whether or not the Activity active on the thread when the **create()** method is called.

marshal_request

Returns a serialized form of the **PropertyGroup** appropriate for propagating within the Activity service context on a request. It is invalid for the parameter to be nil.

marshal_response

Returns a serialized form of the **PropertyGroup** appropriate for propagating within the Activity service context on a response. It is invalid for the parameter to be nil.

unmarshal_request

Returns a reference to a **PropertyGroup** specialization created from the specified serialized form. It is invalid for the parameter to be nil. If the **PropertyGroup** is not known by the importing domain then it is ignored. *pg* is a reference to the **PropertyGroup** context already held by the Activity if it has visited the server previously (in which case the **PropertyGroup** context is being updated rather than created). *parent* is a reference to the **PropertyGroup** parent (if any) so that the **PropertyGroupManager** can ensure correct chaining of nested contexts. The Activity is identified by the gid parameter.

unmarshal_response

This method updates the specified **PropertyGroup** with the specified serialized form received on a response. It is invalid for this parameter to be nil.

destroy

This method is invoked when the **PropertyGroupManager** is no longer required by the Activity service. If the **PropertyGroupManager** has already been destroyed, or is being destroyed, then the CosActivity::AlreadyDestroyed exception will be thrown. Exceptions thrown by *destroy* have no affect on the outcome of an activity.

2.2.9 *CosActivity::Current*

```
interface Current : CORBA::Current
{
   void begin(in long timeout) raises(InvalidState, TimeoutOutOfRange);
   Outcome complete() raises (NoActivity,
         ActivityPending, ChildContextPending, ActivityNotProcessed);
   Outcome complete_with_status(in CompletionStatus cs)
          raises (NoActivity, ActivityPending, ChildContextPending,
             InvalidState, ActivityNotProcessed);

   void set_completion_status (in CompletionStatus cs)
   raises (NoActivity, InvalidState);
   CompletionStatus get_completion_status () raises(NoActivity);

   void set_completion_signal_set (in string signal_set_name)
   raises (NoActivity, SignalSetUnknown);
   string get_completion_signal_set () raises(NoActivity);

   ActivityToken suspend() raises(InvalidParentContext);
   void resume(in ActivityToken at)
   raises (InvalidToken, InvalidParentContext);
```

```
    ActivityToken suspend_all();
    void resume_all(in ActivityToken at)
          raises (InvalidToken, InvalidParentContext);

    GlobalId get_global_id ();

    Status get_status();
    string get_activity_name ();

    void set_timeout (in long seconds) raises(TimeoutOutOfRange);
    long get_timeout ();

    ActivityContext get_context();
    void recreate_context(in ActivityContext ctx) raises(InvalidContext);

    ActivityCoordinator get_coordinator();
    ActivityCoordinator get_parent_coordinator();

    ActivityIdentity get_identity ();
    ActivityToken get_token ();

    PropertyGroup get_property_group(in string name)
    raises(PropertyGroupUnknown, NoActivity);
};
```

The **Activity Current** interface provides operations which allow the demarcation of Activity scope. In addition, it provides interfaces for coordinating the Actions of the current Activity. Once an Activity begins to complete, references to it, or information about it, is no longer available through **Current**. The Activity Service specific **Current** object may be obtained via **resolve_initial_references** with the name "**ActivityCurrent**." As can be seen from the IDL, there are 3 different Current implementations: **CosActivity**'s **Current** is the base **Current**; **CosActivityAdministration**'s **Current** inherits from this; **CosActivityCoordination**'s **Current** inherits from **CosActivityAdministration**'s **Current**. The call to **resolve_initial_references** returns a reference to **CosActivity::Current**, and the application must narrow appropriately to the other **Current** implementations.

Note: some implementations of the service may wish to restrict which implementations of **Current** are available. For example, in a pure client environment, only the **CosActivity::Current** implementation makes sense. Therefore, an implementation need not make all such objects available in all environments and **resolve_initial_references** will behave accordingly.

begin

Creates a new Activity and associates it with the current thread. An instance of a new **PropertyGroup** is also created. If the current thread is already associated with an Activity, the newly created Activity will be nested within it. Otherwise, the Activity

exists at the top level. If the parent Activity has been marked as **CompletionStatusFailOnly**, then the InvalidState exception will be thrown. If it is completing, or has completed, the INVALID_ACTIVITY exception will be thrown.

The *timeout* parameter is used to control the lifetime of the Activity. If the Activity has not completed by the time *timeout* seconds elapses then it is subject to being completed with the **CompletionStatusFail** status. The timeout can have the following possible values:

- *any positive value*: the Activity must complete within this number of seconds.
- *-1*: the Activity will never be completed automatically by the Activity Service implementation (i.e., it will never be considered to have timed out).
- *0*: the last value specified using the **set_timeout** method is used. If no prior call to **set_timeout** has occurred for this thread, or the value returned is 0, then it is implementation dependent as to the timeout value associated with this Activity.

Any other value results in the TimeoutOutOfRange exception being thrown.

complete

Causes the Activity associated with the current thread to complete with its current **CompletionStatus**, or **CompletionStatusFail** if none has been specified using **set_completion_status**. If a registered **SignalSet** has been provided then it will be used for any registered Actions, and they will be invoked appropriately by the Activity's coordinator. If the Activity is nested within a parent, then that parent Activity becomes associated with the thread. If there are any encompassed active or suspended Activities or transactions, and the completion status is **CompletionStatusSuccess**, then ChildContextPending is raised; the application must then either complete the outstanding nested contexts or force the Activity to end by setting the **CompletionStatus** to either **CompletionStatusFail**, **CompletionStatusFailOnly** and then calling *complete* again.

If the completion status is **CompletionStatusFail**, or **CompletionStatusFailOnly**, any encompassed active or suspended Activities will they have their completion status set to **CompletionStatusFailOnly** and transactions will be marked *rollback_only*.

If there is no Activity associated with the current thread, the NoActivity exception is raised and no other action is taken. Only the Activity originator may call **complete()**. The originator is defined as the execution environment in which the Activity is rooted.

If a call to complete the Activity is made from an execution environment into which the Activity was imported, the NO_PERMISSION exception is raised.

If the thread from which the **complete()** call is made is not the only thread on which the Activity is active, then the ActivityPending exception is raised. The application response should be to try again later when any asynchronous work on other threads has been suspended. This method returns an Outcome (or null) which can be used to interpret the final outcome of the Activity.

If no completion **SignalSet** has been set by the application, then the Outcome returned will be null. If the Activity cannot complete in the status required, then the ACTIVITY_COMPLETED exception will be thrown.

If the Activity has begun completion, or has completed, then the INVALID_ACTIVITY exception is thrown.

The ActivityNotProcessed exception is raised in the event that the signals required to complete this operation could not be produced, and the Activity's final completion status is **StatusError**.

complete_with_status

Causes the Activity associated with the current thread to complete and use the **CompletionStatus** provided if this does not conflict with any that has previously been set using **set_completion_status**; this is logically equivalent to calling **set_completion_status** followed by the **complete()** method.

If a registered **SignalSet** has been provided, then it will be used for any registered Actions, and they will be invoked appropriately by the Activity's coordinator.

If the Activity is nested within a parent, then that parent Activity becomes associated with the thread.

If there are any encompassed active or suspended Activities or transactions, and the completion status is **CompletionStatusSuccess**, then ChildContextPending is raised; the application must then either complete the outstanding nested contexts or force the Activity to end by setting the **CompletionStatus** to either **CompletionStatusFail**, **CompletionStatusFailOnly**.

If the completion status is **CompletionStatusFail** or **CompletionStatusFailOnly**, any encompassed active or suspended Activities will they have their completion status set to **CompletionStatusFailOnly** and transactions will be marked *rollback_only.*

If there is no Activity associated with the current thread, the NoActivity exception is raised and no other action is taken. Only the Activity originator may call **complete()**. The originator is defined as the execution environment in which the Activity is rooted.

If a call to complete the Activity is made from an execution environment into which the Activity was imported, the NO_PERMISSION exception is raised.

If the thread from which the **complete_with_status()** call is made is not the only thread on which the Activity is active, then the ActivityPending exception is raised. The application response should be to try again later when any asynchronous work on other threads has been suspended. This method returns an Outcome (or null) which can be used to interpret the final outcome of the Activity.

If no completion **SignalSet** has been set by the application, then the Outcome returned will be null.

If the Activity cannot complete in the status required, then the ACTIVITY_COMPLETED exception will be thrown.

If the Activity has begun completion, or has completed, then the INVALID_ACTIVITY exception is thrown.

The ActivityNotProcessed exception is raised in the event that the signals required to complete this operation could not be produced, and the Activity's final completion status is **StatusError**.

set_completion_status

This method can be used to set the **CompletionStatus** that will be used when the Activity completes. This method may be called many times during the lifetime of an Activity in order to reflect changes in its completion status as it executes.

If this method is not called during the Activity's lifetime, the default status is **CompletionStatusFail**. When the Activity completes, the **CompletionStatus** is given to the registered **SignalSet** (if any) so that it can determine the sequence of Signals to produce.

If the **CompletionStatus** is **CompletionStatusFailOnly** and an attempt is made to change the status to anything other than **CompletionStatusFailOnly**, the InvalidState exception will be thrown. If the Activity has begun completion, or has completed, then the INVALID_ACTIVITY exception is thrown.

get_completion_status

Returns the completion status currently associated with the target Activity. This is the last valid value to **set_completion_status**, or **CompletionStatusFail** if none has been provided.

set_completion_signal_set

This method can be used to set the **SignalSet** that will be used when the Activity completes. If the Activity has begun completion, or has completed, then the INVALID_ACTIVITY exception is thrown.

get_completion_signal_set

Returns the **SignalSet** currently associated with the target Activity that will be used when it completes. This will be the last valid **SignalSet** given to **set_completion_signal_set**, or an empty string if one has not been provided.

suspend

Suspends the Activity associated with the current thread (and any related transactions) and any nested child scopes. An **ActivityToken** representing the Activity that was associated with the current thread prior to this call is returned. The context the handle represents has knowledge of the nested scopes that were active (and also suspended) when the Activity was suspended.

If the current thread is not associated with an Activity or a transaction, then nil is returned from this operation.

If the current thread is only associated with a transaction, then the **ActivityContext** will reflect this.

If the Activity is nested within a parent Activity, then the parent Activity is associated with the current thread, otherwise the current thread has no Activity associated with it.

If the Activity contains transactions and is also nested within another transaction, then the InvalidParentContext exception will be thrown, since it is not possible to suspend only parts of an OTS transaction hierarchy (i.e., the entire transaction hierarchy will be suspended from the invoking thread's context with the result that previously transactional Activities will no longer have transactions within them). The returned **ActivityToken** may be used to *resume* the suspended Activity on any thread but may not be used to *resume_all*.

resume

Resumes the Activity and any nested scopes represented by the **ActivityToken**. The current thread becomes associated with the Activity (or transaction) represented by the token. If the **ActivityToken** does not represent a valid Activity (or is nil), then the InvalidToken exception is raised and no new association is made on the thread. The context into which an Activity is resumed must be the same as the context from which it was suspended, otherwise an InvalidParentContext exception is raised.

suspend_all

Suspends all the scopes (transaction and Activity) associated with the current thread. An **ActivityToken**, representing the entire thread scope structure that was associated with the current thread prior to this call is returned. On completion of this method no Activity or transaction is associated with the thread. The **ActivityToken** returned may be subsequently used on a **resume_all** operation but it may not be used to simply **resume**.

resume_all

Resumes the scopes represented by the **ActivityToken** that must have been previously obtained from a **suspend_all** operation. If the **ActivityToken** does not represent a valid set of scopes (or is nil), then the InvalidToken exception is raised and no new association is made on the thread. If there is currently an Activity or transaction associated with the invoking thread, then the InvalidParentContext exception is raised.

get_token

Returns the **ActivityToken** for the Activity currently associated with the calling thread, or null if there is no associated Activity. This operation returns the token that would be returned if *suspend* had been called (i.e., this token can only be used in a *resume* operation).

get_global_id

Returns the **GlobalId** for the Activity, or nil if there is no Activity associated with the invoking thread.

get_status

Returns the current status of the Activity. If there is no Activity associated with the calling thread, the **StatusNoActivity** value is returned. The effect of this is equivalent to performing the **get_status** operation on the corresponding **ActivityCoordinator** object.

get_activity_name

If there is no activity associated with the calling thread, an empty string is returned. Otherwise, this operation returns a printable string describing the activity. The effect of this request is equivalent to performing the **get_activity_name** operation on the corresponding **ActivityCoordinator** object.

set_timeout

This operation modifies a state variable associated with the target object that affects the time-out period associated with the activities created by subsequent invocations of the *begin* operation which have 0 specified as their timeout value. If the parameter has a non-zero value *n*, then activities created by subsequent invocations of *begin* will be subject to being completed if they do not complete before *n* seconds after their creation. The timeout can have the following possible values:

- *any positive value*: the Activity must complete within this number of seconds.
- *-1*: the Activity will never be completed automatically by the Activity Service implementation (i.e., it will never be considered to have timed out).
- *0*: it is implementation dependent as to the meaning of passing 0 as the value.

Any other value results in the TimeoutOutOfRange exception being thrown.

get_timeout

This operation returns the state variable associated with the target object that affects the time-out period associated with activities created by calls to *begin*. This need not be the time-out period associated with the current Activity, however.

get_context

Returns the **ActivityContext** of the Activity associated with the current thread. Returns null if no Activity is associated with the current thread. The context represents the entire Activity hierarchy (i.e., this operation is equivalent to calling **get_context** on an **ActivityToken** returned by **suspend_all**).

recreate_context

This method can be used by a domain to import from another domain a previously received Activity context. An implementation of the Activity Service which supports interposition uses **recreate_context** to create a new representation of the activity context being imported, subordinate to the representation in *ctx*. If the context cannot be recreated in its entirety (e.g., necessary transaction context information was not propagated as well), or some other failure occurs, then InvalidContext will be thrown.

get_coordinator

Returns a reference to the current Activity's **ActivityCoordinator**. Returns nil of no Activity is associated with the current thread. If an ActivityCoordinator is not supported in this domain then NO_IMPLEMENT will be thrown by the service implementation.

get_parent_coordinator

Returns a reference to the current Activity's parent **ActivityCoordinator**. Returns nil if the current Activity is top-level or no Activity is associated with the current thread.

get_identity

Returns the **ActivityIdentity** for the current Activity, or nil if no Activity is associated with the current thread.

get_property_group

Returns the named **PropertyGroup** for this Activity. If the **PropertyGroup** is unknown, then the PropertyGroupUnknown exception will be thrown. If there is no Activity associated with the calling thread, then the NoActivity exception will be thrown.

2.2.10 CosActivityAdministration::Current

```
interface Current : CosActivity::Current
{
   void register_property_group(in string property_group_name,
                                in PropertyGroupManager manager,
                           in PropertyGroupAttributes attributes)
                                      raises(PropertyGroupAlreadyRegistered);
   void unregister_property_group(in string property_group_name)
                                raises(PropertyGroupNotRegistered);
};
```

register_property_group

Registers the specified **PropertyGoupManager** with the specified name. The Activity Service uses the named **PropertyGroupManager** to create, marshal, and unmarshal **PropertyGroups**. Any top-level Activity started by the invoking thread after this call has succeeded will create an instance of the registered **PropertyGroup**. If the **PropertyGroupManager** has already been registered, then the PropertyGroupAlreadyRegistered exception is thrown.

unregister_property_group

Unregisters the **PropertyGroupManager** with the specified name. Any new top-level Activities started by this thread after the **PropertyGroup** has been unregistered will not create **PropertyGroups** of this type. Existing Activities, or new Activities created as children of existing Activities, are unaffected. If the named **PropertyGroup** is not

known, then the PropertyGroupNotRegistered exception is thrown. **PropertyGroupManagers** must continue to function after they have been unregistered to support Activities that are still using them.

2.2.11 CosActivityCoordination::Current

```
CosActivityCoordination::Current : CosActivityAdministration::Current
{
   CosActivity::Outcome broadcast(in string signal_set_name)
         raises(CosActivity::SignalSetUnknown,
                  CosActivity::NoActivity, CosActivity::ActivityNotProcessed);

   void add_signal_set (in CosActivity::SignalSet signal_set)
                  raises(CosActivity::SignalSetAlreadyRegistered,
                       CosActivity::NoActivity);
   void remove_signal_set (in string signal_set_name)
                  raises(CosActivity::SignalSetUnknown,
                       CosActivity::NoActivity);

   void add_action(in CosActivity::Action act, in string signal_set_name,
              in long priority) raises(CosActivity::SignalSetUnknown,
                                  CosActivity::NoActivity);
   void remove_action(in CosActivity::Action act, in signal signal_set_name)
         raises(CosActivity::ActionNotFound, CosActivity::NoActivity);

   void add_actions(in CosActivity::ActionSeq acts, in string
signal_set_name,
               in long priority) raises(CosActivity::SignalSetUnknown,
                                   CosActivity::NoActivity);
   CosActivity::ActionSeq remove_actions(in CosActivity::ActionSeq acts,
                  in string signal_set_name)
                   raises(CosActivity::NoActivity);

   void add_global_action(in CosActivity::Action act, in long priority)
                  raises(CosActivity::NoActivity);
   void remove_global_action(in CosActivity::Action act)
         raises(CosActivity::ActionNotFound, CosActivity::NoActivity);

   long get_number_registered_actions(in string signal_set_name)
      raises(CosActivity::SignalSetUnknown, CosActivity::NoActivity);
   ActionSeq get_actions(in string signal_set_name)
       raises(CosActivity::SignalSetUnknown, CosActivity::NoActivity);
};
```

add_signal_set

This method registers the specified **SignalSet** with the **ActivityCoordinator**. If the **SignalSet** has already been registered, then the SignalSetAlreadyRegistered exception will be raised. If the **ActivityCoordinator** is in use (i.e., is processing

Signals), or has completed, then the INVALID_ACTIVITY exception is thrown. If there is no Activity associated with the current thread, then the NoActivity exception will be thrown.

remove_signal_set

This method removes the specified **SignalSet** from the **ActivityCoordinator**. If the Activity has begun completion, has completed, or is in the process of using the specified **SignalSet**, then the INVALID_ACTIVITY exception is thrown. If the **SignalSet** is not known, then SignalSetUnknown will be raised. If there is no Activity associated with the current thread, then the NoActivity exception will be thrown. It is invalid to attempt to remove any of the pre-defined **SignalSet**s, and BAD_OPERATION will be thrown.

add_action

Registers the specified Action with the **ActivityCoordinator** such that when the Activity decides to send the specified Signal, the Action will be invoked with that Signal. If multiple Actions are registered, then *priority* may be used to place an order on how they will be invoked: higher priority numbers will be invoked before lower priority numbers. The priority value must be a positive value; a value of zero means that the Activity Service implementation is free to place the Action at any point in the Action list. If the **SignalSet** is not known about, then the SignalSetUnknown exception is thrown. If there is no Activity associated with the current thread, then the NoActivity exception will be thrown. If the Activity has begun completion, or has completed, then the INVALID_ACTIVITY exception is thrown. If the specified Action is registered multiple times for the same SignalSet then it will be invoked multiple times with the Signals from that SignalSet.

add_actions

Registers a number of Actions with the **ActivityCoordinator**; such Actions are assumed to be already prioritized within the sequence. If multiple Actions are registered, then *priority* may be used to place an order on how they will be invoked: higher priority numbers will be invoked before lower priority numbers. The priority value must be a positive value; a value of zero means that the Activity Service implementation is free to place the Action at any point in the Action list. If the **SignalSet** is not known about, then the SignalSetUnknown exception is thrown. If there is no Activity associated with the current thread, then the NoActivity exception will be thrown. If the Activity has begun completion, or has completed, then the INVALID_ACTIVITY exception is thrown. If the specified Action is registered multiple times for the same SignalSet then it will be invoked multiple times with the Signals from that SignalSet.

add_global_action

This method registers the specified Action with the **ActivityCoordinator** such that when *any* Signal is sent, the Action will be invoked with that Signal (i.e., the Action is effectively registering interest in all possible **SignalSet**s). If multiple Actions are registered, then *priority* may be used to place an order on how they will be invoked: higher priority numbers will be invoked before lower priority numbers. The priority value must be a positive value; a value of zero means that the Activity Service implementation

is free to place the Action at any point in the Action list. If there is no Activity associated with the current thread, then the NoActivity exception will be thrown. If the Activity has begun completion, or has completed, then the INVALID_ACTIVITY exception is thrown.

remove_action

Removes the interest relationship between the specified Action and the named SignalSet. No further Signals from the named SignalSet will be sent to the specified Action. If **signal_set_name** is specified as an empty string, then the Action will be sent no further Signals from any SignalSet. If the Action has not previously been registered with the coordinator, then the ActionNotFound exception will be thrown. If there is no Activity associated with the current thread, then the NoActivity exception will be thrown. If the Activity has begun completion, or has completed, then the INVALID_ACTIVITY exception is thrown.

remove_actions

Removes the interest relationship between the specified Actions and the named SignalSet. No further Signals from the named SignalSet will be sent to the specified Actions. If signal_set_name is specified as an empty string, then the Actions will be sent no further Signals from any SignalSet. If any of the Actions have not previously been registered with the coordinator, then it will return references to them after removing all other Actions in the sequence. Otherwise nil will be returned. If there is no Activity associated with the current thread, then the NoActivity exception will be thrown. If the Activity has begun completion, or has completed, then the INVALID_ACTIVITY exception is thrown.

remove_global_action

This method removes the specified Action from the **ActivityCoordinator**. If the Action has not previously been registered with the coordinator, then it will throw the ActionNotFound exception. If there is no Activity associated with the current thread, then the NoActivity exception will be thrown. If the Activity has begun completion, or has completed, then the INVALID_ACTIVITY exception is thrown.

get_number_registered_actions

Returns the total number of Actions that have been registered with the **ActivityCoordinator**. If there is no Activity associated with the current thread, then the NoActivity exception will be thrown.

get_actions

Returns all the Actions that have been registered with the **ActivityCoordinator**. If there is no Activity associated with the current thread, then the NoActivity exception will be thrown.

broadcast

Instructs the **ActivityCoordinator** to send the specified **SignalSet** to all of the registered Actions. Once the Actions have processed the signal and returned outcome Signals, it is up to the **ActivityCoordinator** to consolidate these individual outcomes into a single outcome to return.

If there is no Activity associated with the current thread, then the NoActivity exception will be thrown. This can be used to cause Signals to be sent to Actions at times other than when the Activity completes. As such, the implementation of the Activity Service must ensure that such Signals clearly identify that the Activity is not completing, and that pre-defined **SignalSet**s such as Synchronization, are not used. The result of using the **SignalSet** is returned.

If an attempt is made to use the Synchronization or ChildLifetime **SignalSet**s, then BAD_OPERATION will be thrown and the **ActivityCoordinator** will not be called.

If the Activity has begun completion, or has completed, then the INVALID_ACTIVITY exception is thrown.

The ActivityNotProcessed exception is raised in the event that the signals required to complete this operation could not be produced.

2.2.12 Interposition

When an activity context is propagated, it can be imported by another Activity Service implementation to create a proxy context within the new domain which refers to the exporting domain. This *interposition* technique (supported by the **Current::recreate_context** operation) allows the proxy domain to handle the functions of an Activity Coordinator in the importing domain. These coordinators act as subordinate coordinators.

Interposition allows cooperating Activity Services to share the responsibility for completing an activity and can be used to minimize the number of network messages sent during the completion process. An interposed coordinator registers as a participant in the activity with the **ActivityCoordinator** identified in the **ActivityContext** of the received request; it either registers as an Action, or registers an Action which can then forward Signals to it. The relationships between coordinators in the activity form a tree. The root coordinator is responsible for completing the activity.

A subordinate **ActivityCoordinator** registers itself with its parent as an **Action**, with an interest in the Synchronization **SignalSet**. An Action may be subsequently registered with the subordinate ActivityCoordinator with an interest in a particular SignalSet that is available to the root ActivityCoordinator. The subordinate ActivityCoordinator must have a **SubordinateSignalSet** implementation available to it and should register an Action with an interest in a **SignalSet** of the same name with its superior ActivityCoordinator. When the subordinate ActivityCoordinator receives a **Signal** from its superior it calls the **set_signal** method on the SubordinateSignalSet passing the Signal as a parameter. The subordinate must then forward the Signal to any appropriate Action that registered with it (including other subordinate ActivityCoordinators) and pass each **Outcome** received to the SubordinateSignalSet.

The role of the SubordinateSignalSet is to combine the Outcomes produced into a single Outcome that can be returned to the superior by the subordinate ActivityCoordinator. Once a subordinate ActivityCoordinator has completed distributing a received Signal, it should ask the SubordinateSignalSet for the next signal in case the SubordinateSignalSet is able to produce another Signal, independently of any superior SignalSet, which the subordinate ActivityCoordinator should distribute to any appropriate Actions. Any such Signals are produced as a performance optimization by the SubordinateSignalSet and must not change the Outcome that was produced as a result of the Signal received from the superior.

2.3 Distributing Context Information

The CORE specification must add to the IOP module the following new ServiceId:

```
module IOP
{ // IDL
    const ServiceId ActivityService = 16;
}
```

It is assumed that an appropriate Portable Interceptor will be used to deal with sending and receiving activity context information; this will require the interceptor to un/marshal the context from/into the correct position in the Service Context structure. If Portable Interceptors are not used, then similar mechanisms must be used in order to ensure that context information flows implicitly between execution environments. To ensure interoperability between Activity Service implementations, mechanisms which do not rely upon Portable Interceptors should behave in a similar way to an interceptor and encode the context information appropriately.

It is the responsibility of the Activity Service implementation to register a client and server side interceptor. This is achieved by calling:

- **PortableInterceptor::ORBInitInfo::add_client_request_interceptor(in ClientRequestInterceptor)**
- **PortableInterceptor::ORBInitInfo::add_server_request_interceptor(in ServerRequestInterceptor)**

The interceptor is responsible for marshalling/unmarshalling any Activity context information at the appropriate interception points.

Policing the sending/receiving of Activity context information is dependent on the POA attributes described in the next section.

2.3.1 Activity Service POA Attributes

The Activity Service utilizes a POA policy to define characteristics related to activities. This policy is encoded in the IOR as a tag component and exported to the client when an object reference is created. This enables validation that a particular object is capable of supporting the activity characteristics expected by the client.

```
typedef unsigned short ActivityPolicyValue;

const ActivityPolicyValue REQUIRES = 1;
const ActivityPolicyValue FORBIDS = 2;
const ActivityPolicyValue ADAPTS = 3;
const ActivityPolicyValue INTERNAL = 4;

const CORBA::PolicyType ActivityPolicyType = 58;

interface ActivityPolicy : CORBA::Policy
{
  readonly attribute ActivityPolicyValue apv;
}

const IOP::ComponentId TAG_ACTIVITY_POLICY = 37;
```

ActivityPolicy values are encoded in the TAG_ACTIVITY_POLICY component of the IOR.

The semantics of these policies will now be described (in the following section the term *apv* is the **ActivityPolicyValue** in the Activity component of the target object IOR). Note that an apv of *ADAPTS* should always be treated by a client in the same way as an IOR with no Activity component, in order to work with non-activity aware environments.

Client-side

- If apv is REQUIRES, then a method request must be sent with an Activity context. If there is no Activity context, then the client-side Activity service interceptor must raise the ACTIVITY_REQUIRED system exception and must not send the request.
- If apv is FORBIDS, then no Activity context is allowed to be sent. If there is an Activity context active on the thread, then the client-side Activity service interceptor must raise the INVALID_ACTIVITY system exception and must not send the request.
- If apv is ADAPTS, or if there is no ActivityPolicy, then an Activity context must be sent if and only if an Activity context is associated with the thread of the caller. This would include any requests to objects on a non-Activity aware ORB.
- If apv is INTERNAL then a method request must be sent without an Activity context regardless of whether it is made within the scope of an Activityor not. Activity service implementation objects use this policy.

Server-side

The server-side Activity service interceptor should behave as follows when processing inbound requests:

- If apv is REQUIRES, then any received Activity context must be associated with the thread of execution. If no Activity context is received, the server-side Activity service interceptor must throw the ACTIVITY_REQUIRED system exception, thereby preventing the request from being dispatched.

- If apv is FORBIDS, then the server-side Activity service interceptor is required to check that no Activity context has been flowed with the request and to throw the INVALID_ACTIVITY system exception if it has, thereby preventing the request from being dispatched.
- If apv is ADAPTS, or if there is no **ActivityPolicy**, then any received Activity context must be associated with the thread of execution.
- If apv is INTERNAL, any Activity context must be ignored. The client-side behavior above means that the server should never have to deal with this situation. Given that this situation constitutes a client-side error, an implementation may throw a system exception if this happens.

2.4 The User's View

The following UML diagram briefly illustrates the interactions between the various participants within an Activity during completion.

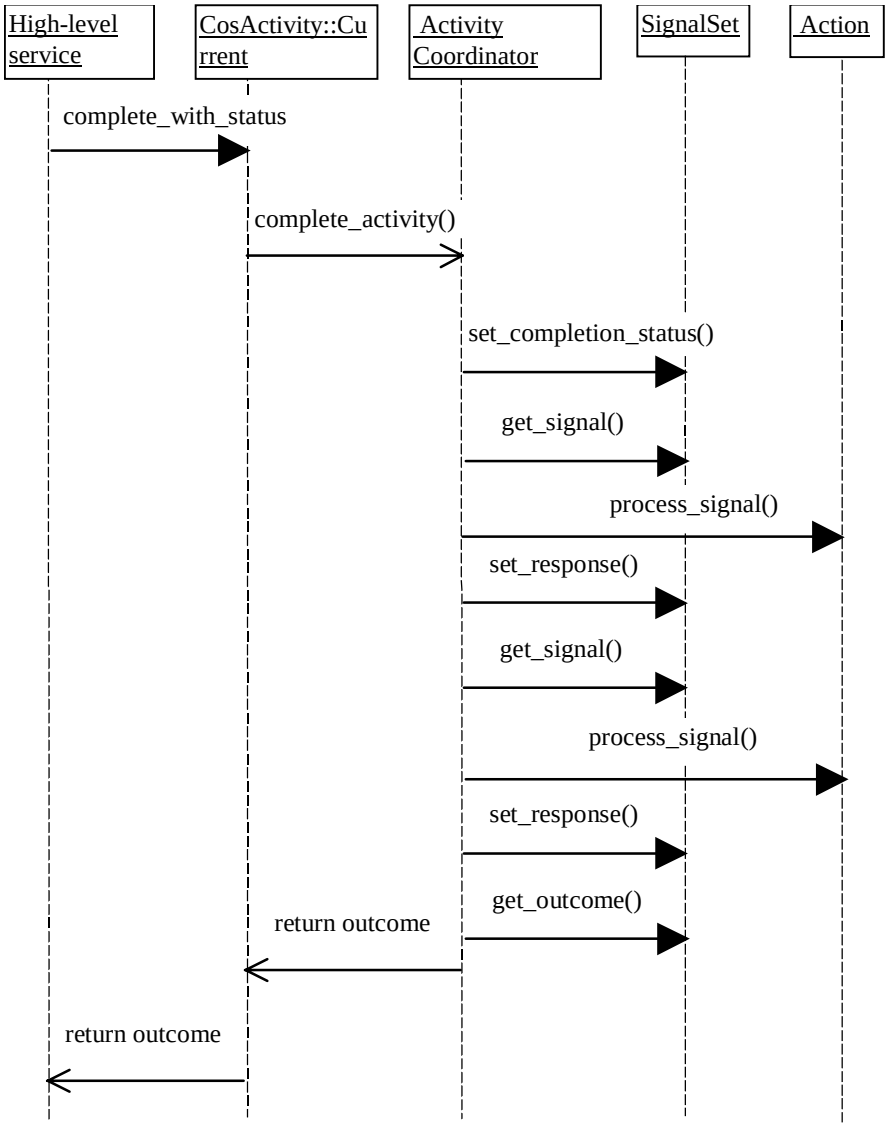

Figure 2-2 Completing an Activity using SignalSets and Actions.

2.4.1 Examples of Use

Using the Activity framework presented previously we wish to provide support for at least the following types of transaction models:

- Workflow-like activities.
- Compensating Activity (Compensating Sphere) with nesting of Activities (spheres) to give recovery behaviour via compensation at all levels of nesting. Support for Sagas as defined in the major section below.

In this section we shall give some brief examples of how these extended forms of transactional activity can be supported. These are meant only as examples, and implementors of the Activity Service framework presented within this specification are not expected to provide them. The Signals and **SignalSet**s described are also meant only as examples.

Concrete examples of specific extended transaction models are provided within Appendix D.

2.4.1.1 Workflow-like Coordination

The signal set required to coordinate the "workflow style" activities contains four signals "start," "start_ack," "outcome," and "outcome_ack."

- *start*: signal is sent from a "parent" activity to a "child" activity (via an Action), to indicate that the "child" activity should start. The **application_specific_data** part of the signal contains the information required to parameterize the starting of the activity. This information is encoded in XML. As noted above, the recipient Action is responsible for starting 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. This information is encoded in XML.
- *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.

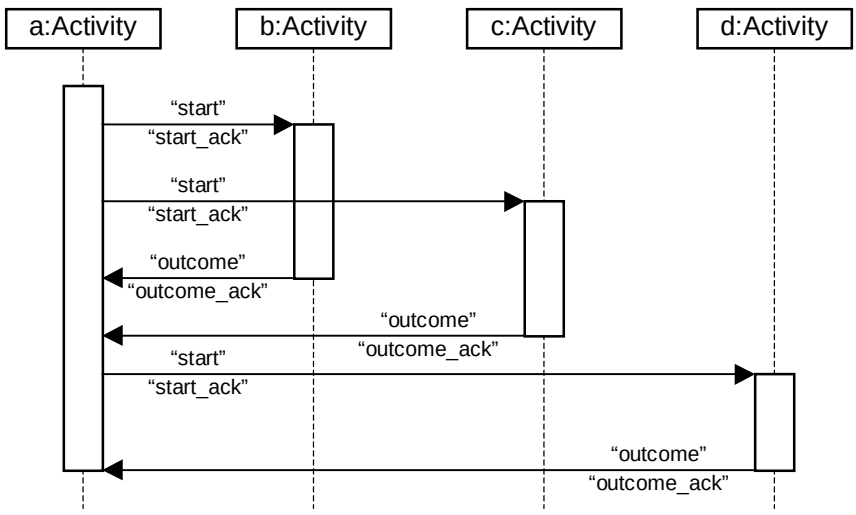

Figure 2-3 Example "Workflow style" activities.

The interaction in Figure 2-3, is activity *a* coordinating the parallel execution of *b* and *c* followed by *d*. For space considerations, the Actions that control the starting of activities b, c and d are not shown, and should be assumed to be implicit in the above diagram.

2.4.1.2 *Compensating Activities*

In this section we shall illustrate how coordination of transactional activities with compensation for failures can be provided using the framework described. Consider the sequence of transactions shown in Figure 1-3 on page 1-5, and assume that each transaction boundary also represents a different activity. The termination of one transaction is used as the driver to start another (perhaps compensating) transaction. We shall assume the existence of a high-level scripting language with which long-running applications can be constructed from short-duration transactions. The signal types required are:

- start: a signal is sent from the terminating activity to the next activity to indicate that it can begin execution. The **application_specific_data** part of the signal contains the information required to parameterize the starting of the activity, such as the state in which this activity has terminated (e.g., committed or rolled back). This information is encoded in XML.
- start_ack: signal is sent from a starting activity to the terminating activity, as the return part of a “start” signal, to acknowledge that the activity has started.

Each activity/transaction may be started by an appropriate Action. Where necessary, the application programmer will be required to implement compensating activities. For example the application programmer must have the necessary knowledge to implement t5(c) which compensates for t2. The application (or some high-level scripting language) will tie together the individual transactional activities such that the ending of one causes the start of another. It is this scripting that will drive the different start signal states in the case of activity failures. For example, if t4 fails then a Signal(start:rolledback) may be sent to t5(c), whereas if t4 completed successfully a Signal(start:ok) may be sent from it to t6.

Sub-activities (sub-transactions) (i.e., activities nested within other activities), would be controlled in a similar manner to the workflow-like scheme presented previously. Compensation would either be left to the enclosing activity or could be handled as described above. If sub-activities are present, then additional signals will be required:

- *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. This information is encoded in XML.
- *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.

2.4.1.3 Two-phase Commit

The UML diagram below illustrates how the Activity Service could be used to implement a two-phase commit protocol, as briefly described in Section 1.2, “Activity Service Model,” on page 1-4. It is assumed that the **process_signal_set** method has been invoked on the **ActivityCoordinator**:

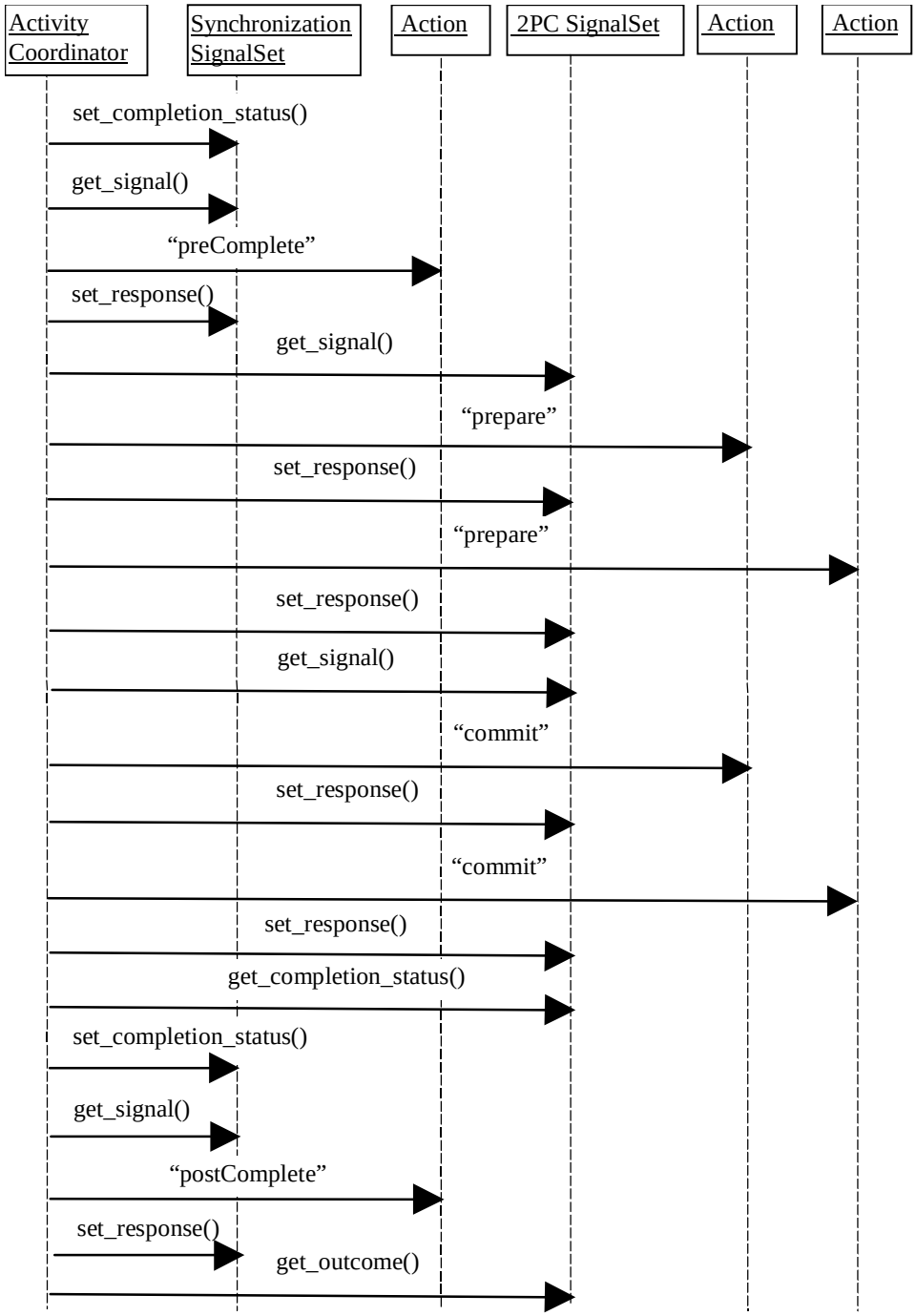

Figure 2-4 Two-phase commit protocol with Signals, SignalSets and Actions.

2.5 The Implementor's View

2.5.1 Suspending Transactions

If **CosTransactions::Current::suspend** is used to suspend a transaction that has nested Activities, then those Activities will not be suspended, since the OTS has no knowledge of Activities. Therefore, we recommend that if such behavior is required, transaction suspending and resuming is performed using the **CosActivity::Current** methods. An implementation of the Object Transaction Service may be made aware of Activities and thus make **CosTransactions::Current** methods respond appropriately. However, this may result in non-portable applications.

2.5.2 Obtaining Current

In order for an application to be able to obtain and use any of the Activity Service Currents it is necessary for an Activity Service to register it with the ORB. The Activity Service implementation is responsible for registering an implementation of the **CosActivityCoordination::Current** as the "ActivityCurrent" returned by **resolve_initial_references**. This is achieved by calling **ORB::register_initial_reference(in ObjectId id, in Object obj)** where **ObjectId** is "ActivityCurrent." Other **Current** implementations may be obtained by suitable narrowing of this object.

2.5.3 Failure Assumptions

Many commercial transaction systems use a *presumed abort* protocol to simply the requirements on failure recovery: if a participant enquires as to the status of a transaction and the system *definitely* has no record about the transaction, then it is assumed to have aborted (rolled back), and the participant can act accordingly. This means that a transaction coordinator need not keep persistent records of participants until after it has decided to commit. Therefore, Activity Service implementations are also required to use a presumed abort (*presumed failed*) protocol.

The Activity Service also assumes that IORs for participants (Actions) and coordinators are persistent, such that upon recovery from failure, an end-point for an IOR remains valid as long as the object it refers to remains in existance. Therefore, a client receiving an OBJECT_NOT_EXIST exception can be guaranteed that the object has ceased to exist because it has successfully completed its job.

2.5.4 Normal Activity Completion

In order to write a portable application or application framework that uses the Activity service, and in order for Activity service implementations to fully interoperate, the ordering and semantics of completion processing of an Activity are described in detail in this section.

1. Current::complete_with_status(comp_status) is called.

2. This drives **ActivityCoordinator::complete_activity(comp_ss_name, comp_status)**. If this is a remote call then no Activity service is marshalled since the target ActivityCoordinator has an ActivityPolicyValue of **INTERNAL**.

3. The *preComplete* synchronization signal is distributed. The Activity context must be available on the thread when the Actions process this signal.

4. The completion signals are distributed to registered Actions. The Activity context must be available on the thread when the completion signals are distributed.

5. The context is logically suspended. Any **PropertyGroups** are called with **suspended()** and then with **completed()**.

6. The *postComplete* synchronization signal is sent.

7. Any remaining Activity service objects for the completing Activity are cleaned up.

8. The call returns to the client.