Additional Structuring Mechanisms for the OTS

This OMG document replaces the draft adopted specification (ptc/01-05-01). It is an OMG Final Adopted Specification, which has been approved by the OMG board and technical plenaries, and is currently in the finalization phase. Comments on the content of this document are welcomed, and should be directed to *issues@omg.org* by November 5, 2001.

You may view the pending issues for this specification from the OMG revision issues web page *http://www.omg.org/issues/*; however, at the time of this writing there were no pending issues.

The FTF Recommendation and Report for this specification will be published on January 24, 2002. If you are reading this after that date, please download the available specification from the OMG formal specifications web page.

OMG Adopted Specification

Additional Structuring Mechanisms for the OTS

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the issue reporting form at *http://www.omg.org/library/issuerpt.htm.*

Contents

Preface

About This Document

Under the terms of the collaboration between OMG and X/Open Co Ltd, this document is a candidate for endorsement by X/Open, initially as a Preliminary Specification and later as a full CAE Specification. The collaboration between OMG and X/Open Co Ltd ensures joint review and cohesive support for emerging object-based specifications.

X/Open Preliminary Specifications undergo close scrutiny through a review process at X/Open before publication and are inherently stable specifications. Upgrade to full CAE Specification, after a reasonable interval, takes place following further review by X/Open. This further review considers the implementation experience of members and the full implications of conformance and branding.

Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

X/Open

X/Open is an independent, worldwide, open systems organization supported by most of the world's largest information system suppliers, user organizations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

Intended Audience

The specifications described in this manual are aimed at software designers and developers who want to produce applications that comply with OMG standards for object services; the benefits of compliance are outlined in the following section, "Need for Object Services."

Need for Object Services

To understand how Object Services benefit all computer vendors and users, it is helpful to understand their context within OMG's vision of object management. The key to understanding the structure of the architecture is the Reference Model, which consists of the following components:

- **Object Request Broker**, which enables objects to transparently make and receive requests and responses in a distributed environment. It is the foundation for building applications from distributed objects and for interoperability between applications in hetero- and homogeneous environments. The architecture and specifications of the Object Request Broker are described in *CORBA: Common Object Request Broker Architecture and Specification.*

- **Object Services**, a collection of services (interfaces and objects) that support basic functions for using and implementing objects. Services are necessary to construct any distributed application and are always independent of application domains.

- **Common Facilities**, a collection of services that many applications may share, but which are not as fundamental as the Object Services. For instance, a system management or electronic mail facility could be classified as a common facility.

The Object Request Broker, then, is the core of the Reference Model. Nevertheless, an Object Request Broker alone cannot enable interoperability at the application semantic level. An ORB is like a telephone exchange: it provides the basic mechanism for making and receiving calls but does not ensure meaningful communication between subscribers. Meaningful, productive communication depends on additional interfaces, protocols, and policies that are agreed upon outside the telephone system, such as telephones, modems and directory services. This is equivalent to the role of Object Services.

What Is an Object Service Specification?

A specification of an Object Service usually consists of a set of interfaces and a description of the service's behavior. The syntax used to specify the interfaces is the OMG Interface Definition Language (OMG IDL). The semantics that specify a services's behavior are, in general, expressed in terms of the OMG Object Model. The OMG Object Model is based on objects, operations, types, and subtyping. It provides a standard, commonly understood set of terms with which to describe a service's behavior.

(For detailed information about the OMG Reference Model and the OMG Object Model, refer to the *Object Management Architecture Guide).*

Associated OMG Documents

The CORBA documentation is organized as follows:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.

- CORBA Platform Technologies
  - *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
  - *CORBA Languages*, a collection of language mapping specifications. See the individual language mapping specifications.
  - *CORBA Services,* a collection of specifications for OMG's Object Services. See the individual service specifications.
  - *CORBA Facilities,* a collection of specifications for OMG's Common Facilities. See the individual facility specifications.

- CORBA Domain Technologies
  - *CORBA Manufacturing*, a collection of specifications that relate to the manufacturing industry. This group of specifications defines standardized object-oriented interfaces between related services and functions.
  - *CORBA Med*, a collection of specifications that relate to the healthcare industry and represents vendors, healthcare providers, payers, and end users.

- *CORBA Finance*, a collection of specifications that target a vitally important vertical market: financial services and accounting. These important application areas are present in virtually all organizations: including all forms of monetary transactions, payroll, billing, and so forth.
- *CORBA Telecoms*, a collection of specifications that relate to the OMG-compliant interfaces for telecommunication systems.

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

<div align="center">

OMG Headquarters
250 First Avenue, Suite 201
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

</div>

Service Description

An increasingly large number of distributed applications are constructed by composing existing applications. The resulting applications can be very complex in structure, and with complex relationships between their constituent applications. Furthermore, the execution of such an application may take a long time to complete, and may contain long periods of inactivity, often due to the constituent applications requiring user interactions. In a distributed environment, it is inevitable that long running applications will require support for fault-tolerance, because machines may fail or services may be moved or withdrawn.

If an application is structured as a collection of transactions, then when executed the application is frequently required to possess some or all of the ACID properties of the individual transactions. However, currently the application programmer has to build application specific mechanisms to do this (such as creating mechanisms for saving application state, creating ad-hoc locking mechanisms, creating mechanisms for compensating transactions and so forth). Therefore, functionality is required for supporting flexible ways of composing an application using transactions, with the support for enabling the application to possess some or all ACID properties. Such support should include facilities for supporting business rules, programming rules, and data usage patterns.

Long-running applications and activities can be structured as many independent, short-duration top-level transactions, to form a "logical" long-running transaction. This structuring allows an activity to acquire and use resources for only the required duration rather than the entire duration of this long-running transactional activity. In the event of failures, to obtain transactional semantics for the entire long-running transaction may require compensation transactions which can perform forward or backward recovery. A transactional workflow system can be used to provide scripting facilities for expressing the composition of these transactions with specific compensation activities where required.

This document attempts to address the above problems with current transaction structuring mechanisms by proposing a low-level architecture for Workflow Engines, Component Management Middleware, and other systems to use to created advanced transaction implementations.

Acknowledgments

The following companies submitted and/or supported parts of this specification:
- International Business Machines
- IONA Technologies Incorporated
- Vertel/Expersoft
- Alcatel
- University of Newcastle upon Tyne
- Bank of America
- INRIA and BULL

Introduction *1*

Contents

This chapter contains the following sections.

Section Title	Page
"Introduction"	1-1
"Activity Service Model"	1-4

1.1 Introduction

A very high level view of the role of the Activity Service is shown in Figure 1-1. An explanation of the terms used in this figure and some rationale underlying the specification's design choices follow.
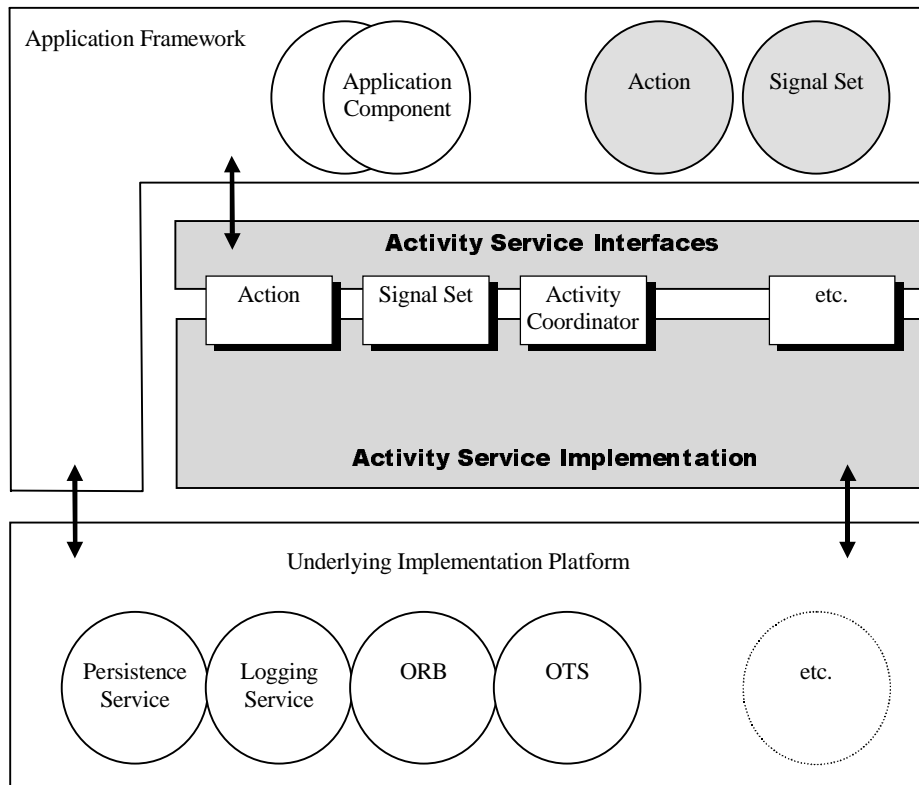
Figure 1-1   Role of the Activity Service

1.1.1 Activity Service Interfaces and Implementation

The behavior required of an Activity Service Implementation which offers the operations in the Activity Service Interfaces is the subject matter of this document. However, it will be useful to understand what an Activity Service Implementation is *not* expected to do by describing the responsibilities and expected behavior of the applications that require it.

1.1.2 Application Framework

It is not expected that the operations in the Activity Services interfaces will be used directly by end-user application programmers. When we talk about application programmers in this document it is really those who write for example, workflow managers or component management systems or who are extending the functionality of the Containers of Enterprise Java Beans (EJBs). Extended transactions like the Sagas and Compensations described below have a complex structure and are intended to last over quite long intervals. Therefore a significant amount of middleware is required to manage the progress and recovery of an extended transaction and is not in the domain of an application programmer who is being employed to write business software rather than middleware.

In this document we have used the term Application Framework to describe the middleware required and to distinguish it from the business application logic of an Activity Component.

It is not the purpose of this document to propose the precise behavior of such workflow management or component management systems -- indeed the OMG has already adopted two related specifications -- rather the purpose is to propose an Activity Service that is needed to allow such Application Frameworks to manage complex *business transactions* that will extend the concept of *transaction* from the well-understood, short-duration atomic transaction of the OMG's Object Transaction service (OTS).

There are expected to be several Application Service Implementations that interoperate in the manner of this proposal and thus will allow extended transactions to span a network of transaction systems connected indirectly by an ORB.

It is one of the responsibilities of an Application Framework implementation to manage the persistent state of its applications; as opposed to the Application Service Implementation'' responsibility to manage the persistent state of its Activity Coordinators and other public and private objects.

1.1.3 Application Component

If the Application Framework is that middleware supplied by specialist vendors required to manage the progress of workflows and long-running business transactions in a variety of business domains, then the term Application Component describes the components that "plug in" to such a framework. The Workflow Management Coalition (WfMC) and the OMG's Workflow Management Facility use the term Activity to describe a step in a path through a workflow digraph. An Application Component -- if it is to be reusable -- has to maintain a degree of independence from the Application Framework in which it runs. Thus a workflow manager might associate one or more Activities with a single Application Component each time giving it different properties that will serve to parameterize the enactment of the workflow [3]. For example, an Application Component that holds a conversation with a graphical user interface (GUI) may be associated with two Activities, one of which has a property "the end-user language is French", the other which has "the end-user language is English," or again, the Application Component could be composed into a larger Activity that must run as a business transactions whose effects can be undone in some situations, whereas it could also be composed into another, or used stand-alone where no transactional behavior is involved at all.

1.1.4 Actions and Signal Sets

This is a specification for an Activity Service that is sufficiently general in its behavior to support a large variety of extended transaction types. As middleware vendors and their customers gain experience in developing and using extended transactions, so more types of extended transactions will emerge.

One of the keys to such extensibility is the Signal Set interface (described in detail below) whose implemented behavior is peculiar to the kind of extended transaction. Similarly, the behavior of an Action will be peculiar to the Application Framework of which it is a part. So as new types of extended transaction emerge, so will new Signal Set instances and associated Actions.

This allows a single Activity Service Implementation to serve a large variety of Application Frameworks, each with its own idea of extended transactions, each with its own Action and Signal Set implementations.

An Activity Service Implementation will not need to know what behavior is encapsulated in the Actions and SignalSets it is given, merely interacting with their opaque interfaces in an entirely uniform and transparent way.

1.1.5 Underlying Implementation Platform

Different Activity Service Implementations will choose to use different combinations of the operating system and transaction services available to them. Although all implementations rely upon the existence of an ORB and an OTS, some implementations will have available to them a transaction system Logging Service and will choose to use it in preference to an Object Persistence Service in order to meet its obligation to recover the state of Activities that persist through failure and restart.

Any dependencies on the functionality of the ORB or Object Services are described below.

1.2 Activity Service Model

1.2.1 Overview

As shown in Figur e1-2, an application activity (shown by the dotted ellipse) is to be split into many different, coordinated, short-duration top-level transactions, to form a "logical long-running transaction."

Application
activity

failure

time →
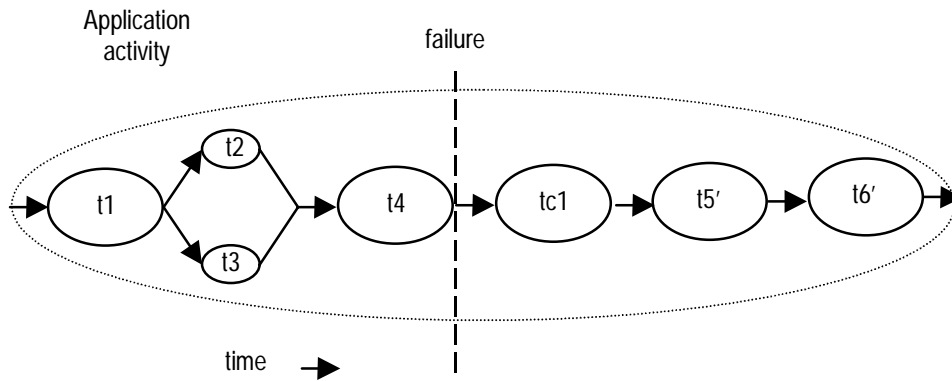
Figure 1-2   An example of a logical long-running "transaction," without failure.

Let us assume that the application activity is concerned with booking a taxi (t1),
reserving a table at a restaurant (t2), reserving a seat at the theatre (t3), and then booking
a room at a hotel (t4).

Application
activity

time →
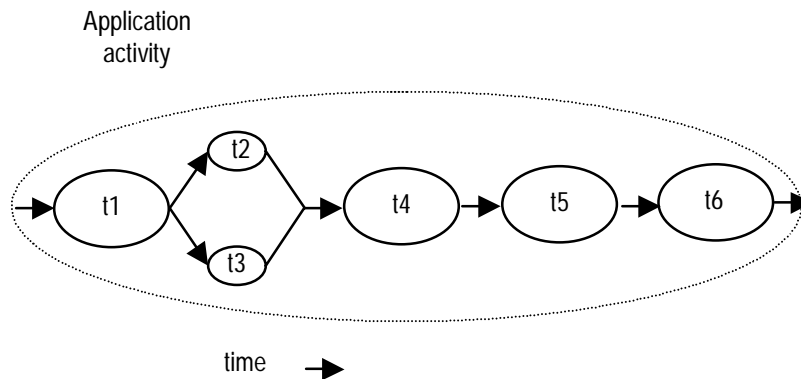
Figure 1-3   An example of a logical long-running "transaction," with failure.

The reason for structuring the application activity as a "logical long-running transaction"
rather than as a single top-level transaction is to prevent certain acquired resources from
being held for the entire duration of the application. It is assumed that the application's
implementors have segmented the transactional activities within the application into
smaller transactions, each such transaction being responsible for acquiring (and
releasing) only those resources it requires.

However, if failures and concurrent access occur during the lifetime of these transactional
activities, then the behavior of the entire "logical long-running transaction" may not
possess ACID properties. Therefore, some form of (application specific) compensation
may be required to attempt to return the state of the system to (application specific)
consistency. For example, consider the above diagrams and let us assume that t4 has
failed (rolls back). Further assume that the application can continue to make forward

progress, but in order to do so must now undo some state changes made prior to the start of t4 (by t1, t2 or t3); since t4 is a transaction, its state changes will be undone automatically by the transaction system, so no form of compensation is required. Therefore, new activities are started; tc1 which is a compensation activity that will attempt to undo state changes performed, by say t2, and t3 which will continue the application once tc1 has completed. tc5' and tc6' are new activities that continue after compensation (e.g., since it was not possible to reserve the theatre, restaurant and hotel, it is decided to book tickets at the cinema). Obviously other forms of transaction composition are possible (e.g., t5' could execute in parallel to tc1).

Much research on structuring applications out of transactions has been influenced by the ideas of *spheres of control* [2]. There are several ways in which some or all of the application requirements outlined above could be met. However, it is unrealistic to believe that the "one-size fits all" paradigm will suffice, i.e., a single high-level model approach to extended transactions is unlikely to be sufficient for all (or even the majority of) applications. Therefore, we propose a low-level infrastructure to support the coordination and control of abstract, application specific entities. As we shall show, these entities (*activities*) may be transactional, they may use weaker forms of serializability, or they may not be transactional at all; the important point is that we are only concerned with their control and coordination, leaving the semantics of such activities to the application programmer.

As we shall show, this distributed coordinator tree will support OTS strict two-phase commit transactions, nested transactions, as well as a variety of different kinds of "transactional behavior" including long-running transactions similar to Sagas with Compensation, Flexible Transactions and Versioning Schemes. Any activity can be associated with issuing demarcation signals (e.g., the end of the saga, the beginning of a compensation group). These signals (or a subset of them) are communicated to any entities that have chosen to register for involvement in the activity context.

An activity is a unit of (distributed) work that may, or may not be transactional. During its lifetime an activity may have transactional and non-transactional periods. Every entity including other Activities can be parts of an Activity, although an activity need not be composed of other activities. An Activity is used to carry transactional and other essential specifications of the application's contract with its middleware.

Each activity is represented by an *activity object*. An Activity is *created*, made to *run*, and then *completed*. The result of a completed activity is its *outcome*, which can be used to determine subsequent flow of control to other activities. Activities can run over long periods of time and can thus be *suspended* and then *resumed* later.

Demarcation signals of any kind are communicated to any registered entities (*actions*) through *signals*. For example, the termination of one activity may initiate the start/restart of other activities in a workflow-like environment. Signals can be used to infer a flow of control during the execution of an application. Actions allow an Activity to be independent of to the specific work it is required to do for signals.

This specification describes basic interfaces that can be used to construct extended transaction models such as long running transactions. These may be used by, for example, workflow engines and applications to do compensation, activity demarcation etc. by sending specific signal information through the Action/Activity structure constructed through the proposed interfaces.

1.2.1.1 Activities and transactions

Note, in the rest of this document when we talk about "transactional activities" we simply mean activities that use (are using, or have used) transactions.

An activity may run for an arbitrary length of time, and may use transactions (and subtransactions) supplied by the Object Transaction Service implementations at arbitrary points during its lifetime. For example, consider Figure 1-4, which shows a series of connected activities co-operating during the lifetime of an application. The solid ellipses represent transaction boundaries, whereas the dotted ellipses are activity boundaries. Activity A1 uses two top-level transactions during its execution, whereas A2 uses none. Additionally, transactional activity A3 has another transactional activity, A3' nested within it.
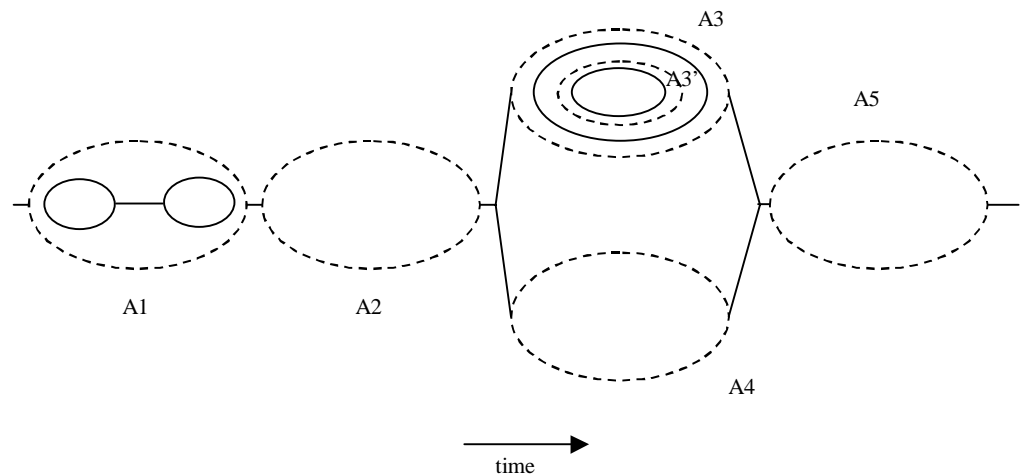
Figure 1-4   Activity and transaction relationship.

Just as a thread of control may require transactional and non-transactional periods and can suspend and resume its transactionality, so too may it require periods of non-activity related work. Thus, it is possible for an activity thread to perform some work outside the scope of the activity before returning to activity related work. In the example diagram above, if the thread performing activity A3' decided to perform some non-activity related work, it could do so outside the scope of A3' and A3. It is not possible to suspend an activity without suspending all of its enclosed transactions. It is desirable that suspending a transaction which has enclosed activities will also suspend those activities. However, this would require that the transaction service has knowledge of activities, and would

require a change to the Object Transaction Service, which is outside the scope of this specification. A possible implementation which does not require changes to the OTS is briefly described in Section 1.2.2, "Coupling Transactions and Activities," on page 1-16.

To ensure application integrity, suspending and resuming of activities implicitly works in conjunction with any transactions that may also be in flight within the activities.

1.2.1.2 Activity Outcome

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

- which are active

- which have completed and what their outcomes were

- which activities failed to complete

This knowledge needs to be related to its own eventual outcome. A responsible entity may be required to handle the sub-activity outcomes; this specification does not mandate how this occurs, but it could be modeled as another Activity so that control flows can be made explicit. The activity determines the collective outcome of the component activity in the light of the various failure and success situations its component activities present it with.

It is outside the scope of this specification to provide interfaces for the outcome manager or to describe how such an entity should be constructed. In the case of complex activity applications, suitable scripting languages may be required to assist the application programmer to define the roles of outcome manager and activities.

1.2.1.3 Activity Failures

The failure of an individual Activity may produce application specific inconsistencies depending upon the type of activity, and how the application depends upon it.

- If the Activity was involved within a transaction, then any state changes it may have been making when the failure occurred will eventually be recovered by the transaction service implementation.

- If the Activity was not involved within a transaction, then application specific compensation may be required.

- An application that consisted of the (possibly parallel) execution of many activities (transactional or not) may still require some form of compensation to "recover" committed state changes made by prior activities. For example, the application shown in Figure 1-3 on page 1-5.

This specification considers that the compensation of the state changes made by an activity is simply the role of another activity; we do not distinguish between compensating and non-compensating activities. A compensating activity is simply performing further work on behalf of the application. Just as application programmers are expected to write "normal" activities, they will therefore also be required to write "compensating" activities, if such are needed. In general, it is only application programmers who possess sufficient information about the role of data within the application and how it has been manipulated over time to be able to compensate for the failure of activities. Automatic compensation of activities may be provided by systems and tools which use the activity framework presented in this specification.

1.2.1.4 Activity Integrity

Activity service implementations must impose constraints on the use of activity interfaces to guarantee integrity. This is similar to the checked transaction behavior of the Object Transaction Service, and an implementation may use similar algorithms to those presented within that specification. Implementations must ensure that all computations acting on behalf of the activity have completed prior to its termination to prevent loss of application integrity. Any enclosed transactions will be responsible for their own integrity checks; hence an Activity Service implementation need only be concerned with imposing checking constraints on work conducted outside the scope of transactions.

1.2.1.5 Signals, SignalSets, and Actions

Activities interact with each other and the rest of the distributed system through *Signals* and *Actions*. An Activity may decide to transmit activity specific data (Signals) to any number of other Activities at specific times during its lifetime (e.g., when it terminates); each signal may be used to represent the requested outcome of the Activity. The receiving Activities may either have been running and are waiting for a specific Signal, or started by the receipt of the Signals.

To allow Activities to be independent of the other activities, and to allow the insertion of coordination and control points within an application which are outside of the domain of an Activity, Signals are sent to *Actions*, rather than Activities. (The role of an Action is similar that of the **CosTransactions::Resource** in the OTS.) An Action may then use the information encoded within the Signal in an application specific manner. When the Action has finished it may return an application specific indication of the outcome of its having dealt with the Signal. An Action may be considered as an entry/exit point into/from an Activity. However, this specification does not restrict the role of the Action and this should be considered as an example of usage only.
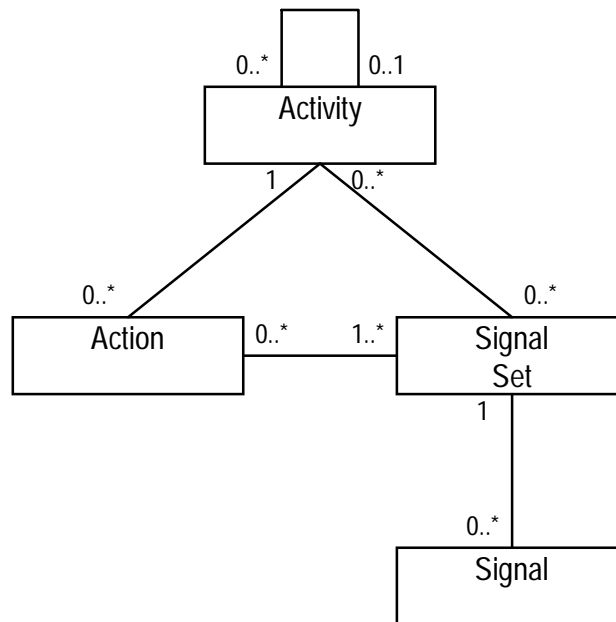
Figure 1-5   Relationship of SignalSets, Signals, Actions and Activities.

To allow Actions to be selectively signaled, Signals are associated with SignalSets, and Actions are implicitly associated with SignalSets. When a Signal is raised it does so within the context of a specific SignalSet, and only those Actions registered with that SignalSet will receive the Signal. An Action may register interest in more than one SignalSet and an Activity may use more than one SignalSet during its lifetime.

An *activity coordinator* may be implicitly associated with each activity, and is used to drive the Signal and Action interactions: if an Activity has no associated Actions, then it need not have an activity coordinator. Activities that require to be informed when another Activity sends a specific Signal can register an appropriate Action with that Activity's coordinator. When the Activity sends a Signal (e.g., at termination time), the coordinator's role is to forward this to all registered Actions and to deal with the outcomes generated by the Actions by passing them to the SignalSet; it is the SignalSet's responsibility to collate the many individual outcomes into a single outcome for the Activity.

With the exception of some predefined Signals and SignalSets, the majority of Signals and SignalSets will be defined and provided by the higher-level applications that make use of this Activity Service framework. To use the generic framework provided within this specification it is necessary for these higher-level applications to impose application specific meanings upon Signals and SignalSets (i.e., to impose a structure on their abstract form). A Signal with the name "foobar" can mean one thing when used within one application, but the same name may have a completely different meaning when used elsewhere.

A SignalSet is responsible for determining which Signals should be sent to registered Actions. The set of Signals a given SignalSet can generate may change from one use to another, and the actual set of Signals it sends may be a subset of these Signals. The intelligence about which Signal to send to an Action is hidden within a SignalSet and may be as complex or as simple as is required by the activity implementation. When a Signal is sent to an Action, the SignalSet is informed of the Outcome generated by that Action to receiving and acting upon that Signal; the SignalSet may then use that information when determining the nature of the next Signal to send. When a given Signal has been sent to all registered Actions the SignalSet will be asked for the next Signal to send by the Activity Coordinator. It is possible for the outcome of an Action to cause the premature fetching of a new Signal from a SignalSet such that not every registered Action will see all of the Signals the SignalSet produced.

When all Signals have been generated by the SignalSet, the Activity's final Outcome can be obtained from the SignalSet. Since all of the Outcomes returned by each Action (including failure Outcomes) have been passed to the SignalSet, it has the responsibility for determining the final Outcome for the Activity. Only the SignalSet has the necessary semantic information to interpret each Outcome in order to make this determination.

As shown below, a given SignalSet is assumed to implement a state machine, whereby it starts off in the *Waiting state* until it is required by the Activity Coordinator to send its first Signal, when it then either enters the *Get Signal state* or the *End state* if it has no Signals to send. Once in the *End state* the SignalSet cannot provide any further Signals and will not be reused. Once in the *Get Signal* state the SignalSet will be asked for a new Signal until it enters the *End state*. A new Signal is only requested from the SignalSet when all registered Actions have been sent the current Signal, or an exceptional outcome is generated by an Action.
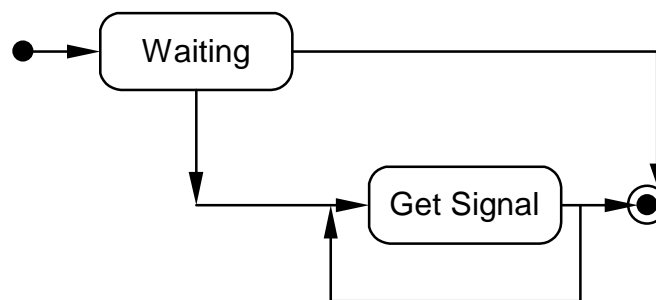
Figure 1-6   SignalSet state transition diagram.

For example, suppose we have a TwoPhaseCommit SignalSet to represent the termination protcool for a transaction, and register Actions with the Activity as the transactional resources; as with the OTS, it is up to the users of the Activity Service to ensure that appropriate Actions are registered at appropriate times.

activity coordinator

transmit signal

get signal

action   action   action   action
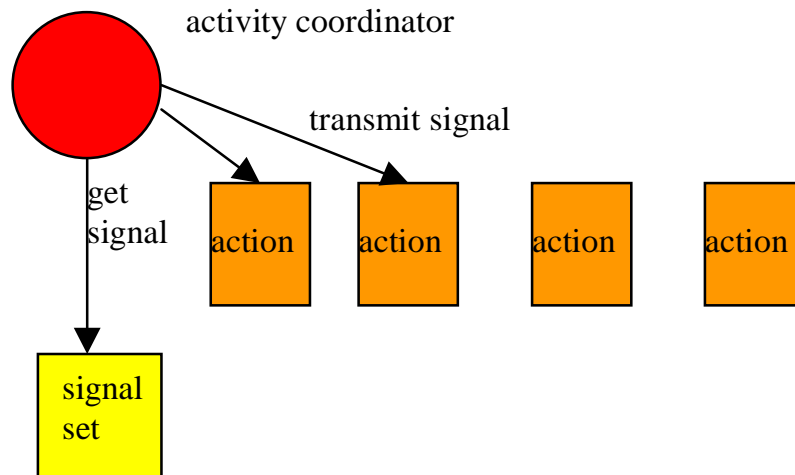
signal set

Figure 1-7   Activity coordinator signaling actions.

When the activity is told to complete, it will be in either a "success" or "failure" state, and this will be communicated to the SignalSet associated with it. If the state is "failure", then the SignalSet would generate a "rollback" Signal, whereas if "success" the SignalSet would generate the "prepare" Signal to be sent to the registered Actions, as shown in Figure 1-7. The Activity Coordinator would then send this Signal to each Action, and inform the SignalSet of the result. Assuming none of the Actions returns an exceptional response to this Signal, then when all Actions have received the "prepare" Signal, and the Activity Coordinator asks the SignalSet for the next Signal, it will return the "commit" Signal. However, if during the "prepare" phase, an Action returns a response which indicates that there is no point in sending the "prepare" Signal to further Actions, the Activity Coordinator will be required to obtain a new Signal from the SignalSet (the "rollback" Signal in this case), and send this to all registered Actions. As stated previously, the intelligence about which Signal to send, and about interpreting outcomes from Actions, resides within the SignalSet, allowing implementations of the framework presented within this specification to be highly configurable, to match application requirements.

The desired delivery semantics for Signals are at least once, although implementations are free to provide better deliver guarantees. This means that an Action may receive the same Signal from an Activity multiple times, and must ensure that such invocations are idempotent (i.e., that multiple invocations of the same Signal to an Action are the same as a single invocation).

1.2.1.6 Contexts

Whenever an entity within an Activity performs an operation it does so within that Activity's context. Since Activities may be nested, the context may form a hierarchy. Because it is important that any operation executes within the correct context, whenever an entity invokes a remote operation on another entity it is necessary to transmit this context information between distributed entities.

Therefore, as part of the environment of each ORB-aware thread, the ORB maintains an activity context; if the activity is transactional then this activity context will have knowledge of the relevant transaction context. The activity context is either null (indicating that the thread has no associated activity) or it refers to a specific activity. It is permitted for multiple threads to be associated with the same activity at the same time, in the same execution environment or in multiple execution environments.

This specification implicitly assumes that transaction context information will be propagated implicitly between execution environments. Although it is assumed that Portable Interceptor technology will be used to accomplish this, this is not mandated, and any similar mechanism may be used; a subsequent section will describe in detail the role required from an interceptor or its equivalent. However, it is assumed that for inter-operability purposes such mechanisms will work with implementations that do use Portable Interceptors. Additional POA attributes have been declared to provide a flexible way in which the activity context may flow between execution environment. These attributes will be described later in the specification.

1.2.1.7 Properties

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

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

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

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

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

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

An Activity can support any number of registered PropertyGroups, each with its own set of behavior. In general it is desirable for all PropertyGroups to display the same behavior with respect to nested Activities, although this is not required. That behavior should be that, when a nested Activity is begun, all of the parent's properties are still available and may be updated, but that updates are scoped to the Activity in whose scope they are made. Thus, when a nested Activity is completed, the parent PropertyGroup is restored to the state it had prior to the nested Activity starting.

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

Any number of named PropertyGroups may be registered with the Activity Service. When an Activity is begun, an instance of each registered PropertyGroup is created and initialized.

Note, an implementation of a PropertyGroup may use an implementation of the OMG's Property Service specification.

1.2.1.8 Recovery

Recovering applications after failures, such as machine crashes or network partitions, is an inherently complex problem: the states of objects in use prior to the failure may be corrupt, and the references to objects held by remote clients may be invalid. At a minimum, restoring an application after a failure may require making object states consistent. The advantage of using transactions to control operations on persistent objects is that the transactions ensure the consistency of the objects, regardless of whether or not failures occur. A transaction system (e.g., one based upon the Object Transaction Service), will guarantee that in the event of failures, any transactions that were in flight will either be committed or rolled back, making permanent or undoing any changes to objects that had occurred (i.e., it will be as though the transactions either did not start, or completed as required).

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

In a similar way, we do not want to mandate a specific persistence and recovery mechanism for the Activity Service. Rather we wish to state what the requirements are on such a service in the event of a failure, and leave it to individual implementors to determine their own recovery mechanisms. As far as end-users of the Activity Service and its applications are concerned, recovery is something that happens after a failure: how it happens is rarely of concern. In a distributed application, where an individual activity may run on different implementations of the Activity Service during its lifetime, recovery is the responsibility of these different implementations. Each implementation may perform recovery in a completely different manner, forming *recovery domains*. Therefore, we are more concerned with the requirements that the Activity Service places on recovery, rather than how such recovery occurs.

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

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

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

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

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

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

If Activities and transactions co-operate within a given application, then the respective recovery mechanisms will also be required to co-operate. Obviously it is not necessary for a user of the Activity Service implementation to use transactions at all, in which case only Activity recovery will be required in the event of a failure (i.e., it is possible to have recovery domains that do not require a transaction service implementation at all).

1.2.2 *Coupling Transactions and Activities*

It is possible for an Activity Service implementation to terminate (or mark for termination) both Activities and transactions because it has sufficient knowledge of both entities. However, without modifying the Object Transaction Service specification, the information about Activities executing within a transaction is not available to a terminating transaction; hence such a transaction cannot force the termination of any sub-activities, only sub-transactions.

It is important to enable the termination of a transaction to affect any enclosed Activities. This specification therefore requires that there be a uniform architectural model, whereby Activities and transactions can cause the termination of *any* enclosing Activities and transactions. Although some implementations of the Object Transaction Service may do this through proprietary extensions to their implementations, this is not strictly necessary. Activities may register a suitable **CosTransactions::Resource** to act on their behalf with any enclosing transaction upon creation; this Resource will be informed whenever the transaction terminates, and the Activity can then behave accordingly (e.g., force the transaction to rollback if the state of the Activity is not consistent).

Modules and Interfaces   *2*

Contents

This chapter contains the following sections.

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.

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.

If the parent of a sub-Activity is not a root Activity (i.e., it is an interposed subordinate) then the distribution of the ChildLifetime signals is delegated upstream to the superior **ActivityCoordinator**.

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**.

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.

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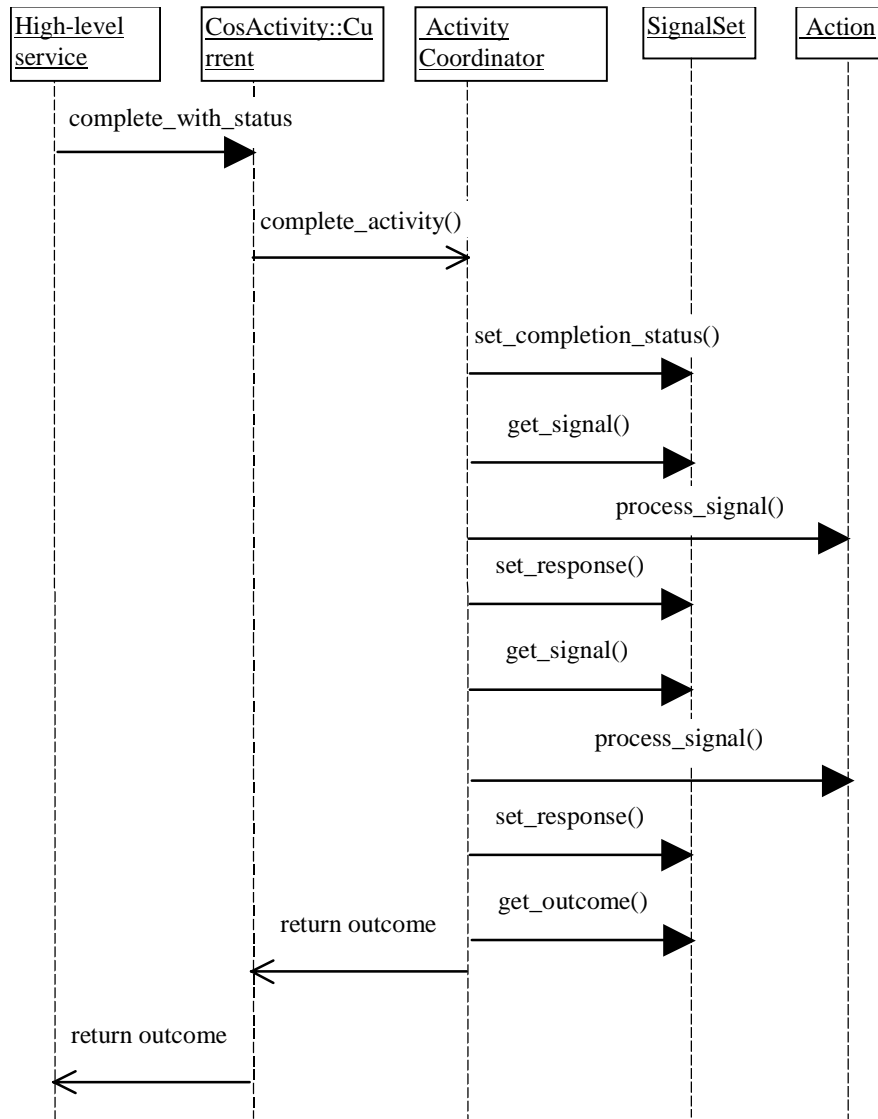
Figure 2-2   Completing an Activity using SignalSets and Actions.

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.
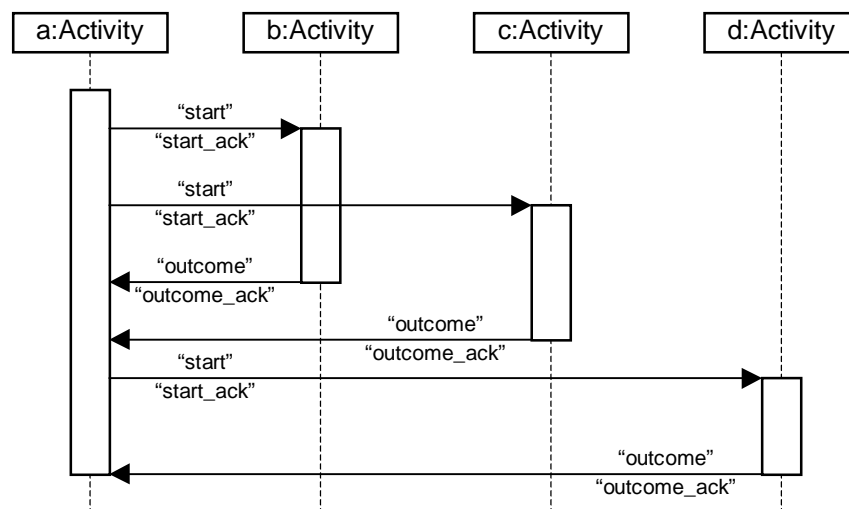

Figure 2-3   Example "Workflow style" activities.

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**:

| Activity Coordinator | Synchronization SignalSet | Action | 2PC SignalSet | Action | Action |

set_completion_status()

get_signal()

"preComplete"

set_response()

get_signal()

"prepare"

set_response()

"prepare"

set_response()

get_signal()

"commit"

set_response()

"commit"

set_response()

get_completion_status()

set_completion_status()

get_signal()
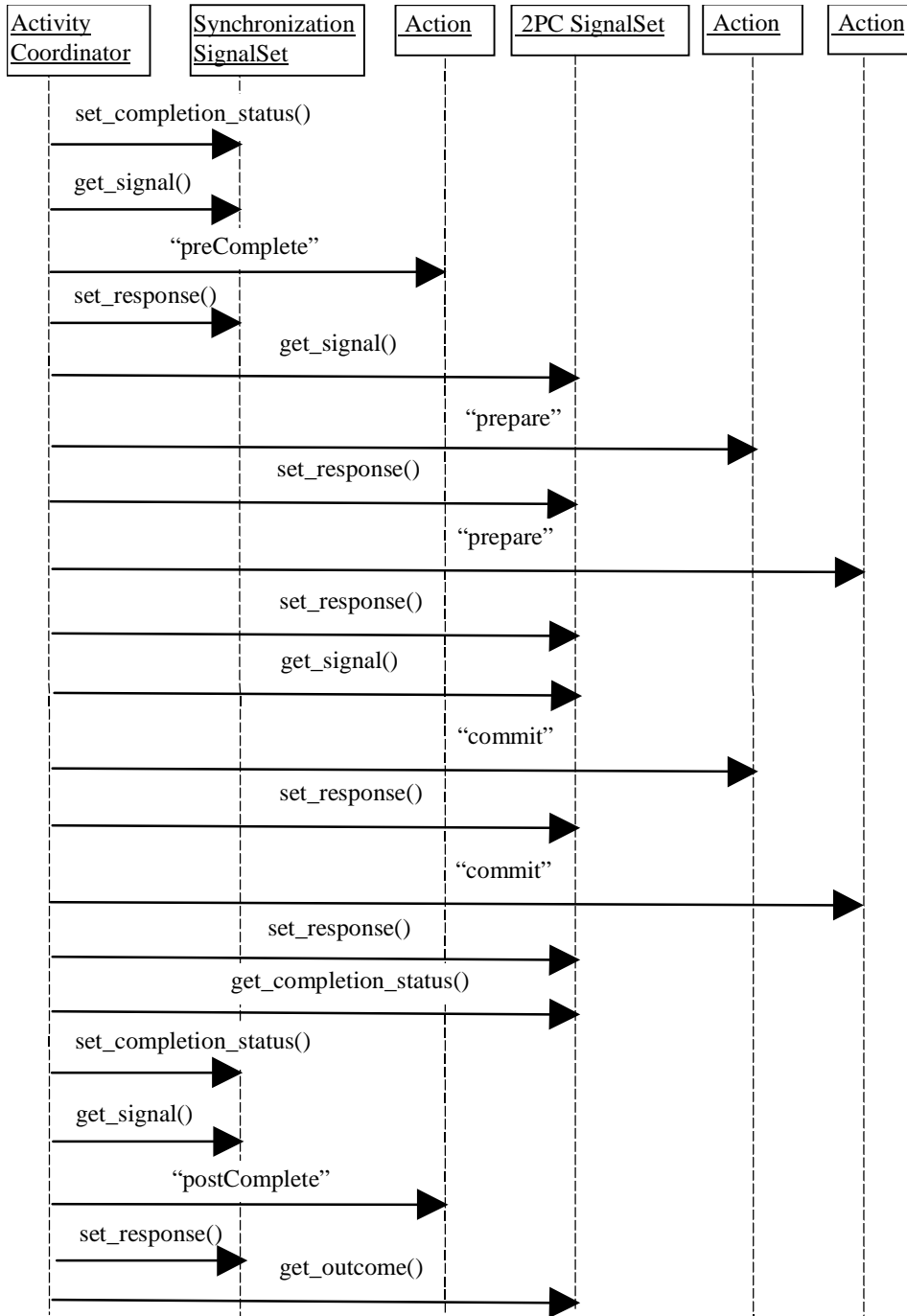
"postComplete"

set_response()

get_outcome()

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.

References *A*

A.1 List of References

1. R. Soley (ed.), Object Management Architecture Guide, Third Edition, Wiley, June 1995.

2. C. T. Davies, "Data processing spheres of control", IBM Systems Journal, Vol. 17, No. 2, 1978, pp. 179-198.

3. J. J. Halliday, S. K. Shrivastava, and S. M. Wheater, "Implementing Support for Work Activity Coordination within a Distributed Workflow System", Proceedings of the Third International Conference on Enterprise Distributed Object Computing (EDOC '99), September 1999, pp. 116-123.

OMG IDL　　　　　　　　　　　　　　　　　　　*B*

B.1   Complete IDL Listing

```
// File: CosActivity

#ifndef COSACTIVITY_IDL_
#define COSACTIVITY_IDL_

#include <orb.idl>

#pragma prefix "omg.org"

module CosActivity
{
  exception NoActivity {};
  exception ActivityPending {};
   exception ActivityNotProcessed {};
  exception InvalidToken {};
  exception InvalidState {};
  exception InvalidContext {};
  exception ActionError {};
  exception AlreadyDestroyed {};
  exception ActionNotFound {};
  exception ChildContextPending {};
  exception InvalidParentContext {};
  exception SignalSetUnknown {};
  exception SignalSetAlreadyRegistered {};
  exception SignalSetActive {};
  exception SignalSetInactive {};
  exception TimeoutOutOfRange {};
  exception PropertyGroupUnknown {};

  interface ActivityCoordinator;
```

```
// The following system exceptions are added to support the Activity
// service

//  INVALID_ACTIVITY
//  ACTIVITY_COMPLETED
//  ACTIVITY_REQUIRED

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;

  typedef sequence<octet> GlobalId;

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

  enum CompletionStatus
  {
    CompletionStatusSuccess,
    CompletionStatusFail,
    CompletionStatusFailOnly
  };

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

  struct Outcome
  {
    string outcome_name;
```

```
            any application_specific_data;
         };

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

       struct PropertyGroupIdentity
       {
          string property_group_name;
          any context_data;
       };

       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;
       };

       interface PropertyGroup
       {
          readonly attribute string property_group_name;

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

          void destroy() raises(AlreadyDestroyed);
       };

       interface SignalSet
       {
          readonly attribute string signal_set_name;

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

          boolean set_response (in Outcome response, out boolean nextSignal)
```

```
raises (SignalSetInactive);

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

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

    void destroy() raises(AlreadyDestroyed);
};

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

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

    void destroy() raises(AlreadyDestroyed);
};
typedef sequence<Action> ActionSeq;

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) raises(ActionNotFound);

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

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(SignalSetUn-
known);

        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);
    };

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

    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, InvalidState);
      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);
        };
    };

    module CosActivityAdministration
    {
        exception PropertyGroupAlreadyRegistered {};
        exception PropertyGroupNotRegistered {};
        exception AttributeAlreadyExists {};
        exception NoSuchAttribute {};

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

        interface PropertyGroupManager
        {
            CosActivity::PropertyGroup create(in CosActivity::PropertyGroup par-
    ent,
                        in CosActivity::GlobalId gid);

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

            CosActivity::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);
  };

  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);
  };
};

module CosActivityCoordination
{
  interface 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::NoActiv-
ity);
    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)
  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)
      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);
        CosActivity::ActionSeq get_actions(in string signal_set_name)
          raises(CosActivity::SignalSetUnknown, CosActivity::NoActivity);
    };
};

#endif
```

Glossary *C*

C.1 Activity Service Terms

Action. When an Activity require Signal processing, Actions will be invoked with the specified Signal.

Activity. An activity is a unit of (distributed) work that may, or may not be transactional. During its lifetime an activity may have transactional and non-transactional periods.

ActivityCoordinator. The coordinator is responsible for coordinating the interactions between Activities through Signals and Actions.

Activity Context. The activity information associated with a specific thread.

Child activity. An activity that has been created within the scope of another activity.

Compensation. An activity that can be used to return the state of the system to application specific consistency.

Current. This interface provides operations which allow the demarcation of Activity scope. In addition, it provides interfaces for coordinating the Actions of the current Activity.

Parent activity. An activity that has child activities.

PropertyGroup. A tuple-space for specifying application specific logic for coordinating and controlling the behavior of activities.

Recovery. A series of actions for restoring the state of the system to application specific consistency.

Root activity. An activity that does not occur within the scope of another activity.

Sibling. A child activity.

C

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

SignalSet. Each Signal is associated with a specific SignalSet. A SignalSet represents the set of Signals that are required to achieve some goal. For example, an OTS SignalSet may contain prepare/commit/rollback/forget/commit_one_phase Signals. Actions register interest in receiving Signals from a given SignalSet.

Transaction. An Object Transaction Service transaction.

Specific Models D

D.1 *Examples of Extended Transaction Models*

In the first part of this document we presented a general framework for the construction of arbitrary extended transaction models. By itself this framework does not present a specific type of extended transaction, and is not intended to be used directly: as previous sections have attempted to explain, it is envisioned that this layer can be provided for, and used by, specific extended transaction models and their implementations.

In this appendix we present some of these extended transaction models, with their own IDL, and illustrate how implementations of these models use the underlying generic framework presented earlier. Note, these extended transaction models are presented as illustration only, and an implementation of this specification need not provide implementations of these models. They are neither mandatory nor optional for a conformant implementation of the Activity Service.

D.1.1 *The Open Nested Transactions Model*

The concept of a transaction has been developed to permit management of activities and resources in a reliable computing environment. Indeed, transactions are useful to guarantee consistency of applications even in case of failure and in the case of conflicting concurrent applications. The traditional or flat transaction model, implied by the OMG (Object Management Group) Object Transaction Service (OTS), although suitable for applications using short transactions, may not provide enough flexibility and performance when used for more complex applications, such as CAD applications, connection establishment in telecommunication or business travel including several servers on different sites and need access to many resources involved within a relatively long-lived transaction.

Typically, the two-phase commit (2PC) protocol is combined with the strict two-phase locking protocol, as the means for ensuring atomicity and the serializability of transactions. The implication of this combination on the length of time a transaction may holding locks on various data items might be severe. At each site, and for each

transaction locks must be held until either a commit or an abort message is received from the coordinator of the 2PC protocol. Since the 2PC protocol is a blocking protocol, the length of time these locks are held can be unbounded.

There are certain classes of application where it is known that resources acquired within a transaction can be "released early," rather than having to wait until the transaction terminates. These applications share a common feature that application-level consistency is maintained, despite any non-ACID behavior they may exhibit. For some applications, failures do not result in application-level inconsistency, and no form of compensation is required. However, for other applications, some form of compensation may be required to restore the system to a consistent state from which it can then continue to operate.

In this section we describe how the "*Open Nested Transaction Model*" (ONT), or the Nested top-level transactions with compensation may be provided using the Activity Service Framework. The Open Nested Model improves greatly transaction parallelism by releasing the nested transaction locks at the nested transaction commit time. That is, open nested transactions relax the isolation property by allowing the effects of the committed nested transaction to be visible to concurrent transactions, thus waiving the lock transfer rule of closed nested transactions.

Since the Activity Service proposed by this specification proposes a low-level architecture to create an advanced transaction model, it appears judicious to provide for end-users wishing to use a particular advanced model a high level API which hides the way the Activity Service is used to provide that advanced model. For this aim we provide, in this submission, an API which allows users to develop transactional activities structured in a hierarchically way reflecting the Open nested transaction model.

The Transactional Model

In this model an Activity may contain any number of nested activities, which may recursively contain other nested activities organized into a hierarchical tree of nested activities or an *Activity family*. In the earlier part of this specification the notion of an Activity was defined in a loose manner, to enable specific extended transaction models to refine what they mean by Activity. The notion of Activity used within this section is therefore specific to this model, and should not be confused with any other Activity definition used by other extended transaction models.

Each activity or nested activity represents an atomic unit of work to be done; that is an OTS transaction. The creation of an activity or nested activity implies the creation of an associated top-level, or flat transaction, which may possibly contain nested transactions, if the provided Object Transaction Service supports nested transactions. That is, from the application point of view, an activity is implicitly transactional. Therefore, unless otherwise stated, in the rest of this section we shall use the term Activity to refer to the Activity and its associated transaction; operations which are applied to the Activity are likewise assumed to be applied to the transaction where appropriate, in order to guarantee consistency.

The transaction model respects the following rules:

- Sub-activities are strictly nested. An Activity or Sub-activity cannot complete with Success unless all of its children have completed. Since an activity is implicitly transactional, completing with Success means that the associated transaction is committed.

- When an Activity or Sub-activity completes with Failure, all of its children in an active state are completed with Failure. Since an Activity is implicitly transactional, completing with Failure means that the associated transaction is rolled back.

- When an Activity or Sub-activity complete with Failure or rolls back, all of its children which have completed with Success or committed shall be compensated if compensating actions have been defined. The behavior of the compensation action is defined by the application since it is only the application that possesses sufficient information to do compensation.

The end-user programming interface

To avoid using more than one interface to manage an Activity and a transaction within an activity, the interface provided for end-users to create ONT transactional activities relies on a new Current interface. This interface invokes the appropriate interfaces provided by the underlying Transaction Service and Activity Service, respectively, to manage transactions and to manage activities.

For simplicity, the intermediate mechanism allowing to manage Open Nested transactions and located between the end-user applications and the Activity Service is referred to as the *OpenNested Service*.

Although a high level Current interface is added to hide those provided by the Activity Service and the Transaction Service, we do not mandate a new context to be propagated among participants within a same transactional activity. The context to be propagated relies on the policy defined by the invoked object as explained in Section 1.2.1.6, "Contexts," on page 1-13.

Datatypes

enum Activity_Status {
    StatusActive,
    StatusNoActivity,
    StatusMarkedRollback,
    StatusRollingBack,
    StatusCommitting,
    StatusRolledBack,
    StatusCommitted,
    StatusToCompensate,
    StatusUnknown
};

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

- **StatusActive**: An Activity is associated with the target object. The Activity and its associated transaction is in the active 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.

- **StatusMarkedRolledback**: The transaction associated with the target object or the target activity has been marked for rollback. The activity will complete with the CompletionStatusFail.

- **StatusRollingBack** : A transaction is associated with the target object and it is in the process of rolling back. An implementation returns this status if it has decided to rollback, but has not yet completed the process because it is waiting for responses from the Transaction Service.

- **StatusCommitting**: The transaction or Activity associated with the target object is in the process of committing. An implementation returns this status if it has decided to commit, but has not yet completed the process because it is waiting for responses from the Transaction Service.

- **StatusRolledback**: An Activity is associated with the target object and it has completed with the status CompletionStatusFail or CompletionStatusFailOnly and its associated transaction has rolled back.

- **StatusCommitted**: An Activity is associated with the target object, it has completed with the status CompletionStatusSuccess and its associated transaction has committed. There is no Compensation defined for that activity.

- **StatusToCompensate**: An Activity is associated with the target object, it has completed with the status CompletionStatusSuccess and its associated transaction has committed. A Compensation has been defined for that activity and is waiting for its ancestors' outcome.

- **StatusCompleted**: An Activity is associated with the target object and it has completed. That is either it has committed and a compensation has not been defined, or it has rolled back, or it has been compensated.

- **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.

The diagram below indicates the transitions a transactional Activity can undergo. Because the interfaces described in the first part of this specification are meant to define a generic framework for many extended transaction models, the Activity statues described in Section 2.1.2.2, "Status," on page2-2 do not convey fine-grained knowledge about the application-level progress of an Activity; such information is not available at the level of those interfaces since the concept of an Activity depends somewhat upon the application semantics. However, at the level of the ONT interfaces, the notion of an Activity is tied to the ONT model, and finer granularity statuses can be given to the application to indicate the transactional Activity's progress. Obviously the low-level statuses provided by the general framework are available to the application if it requires them.
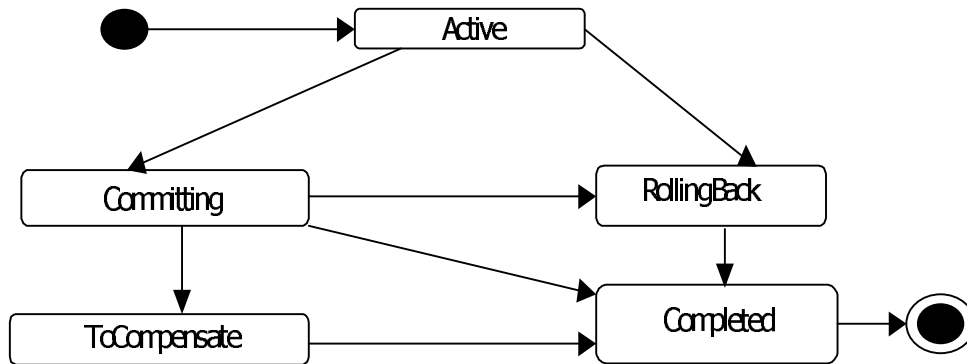
Figure D-1   Transactional Activity and UML state diagram.

Exceptions

We define the following exceptions:

exception Heuristic_Compensate {};
exception Heuristic_No_Compensate {};
exception Activity_RolledBack {};

Heuristic_Compensate Exception

The Heuristic_Compensate exception is raised to report that a compensation has been performed while the entire activity has been requested to commit.

Heuristic_No_Compensate Exception

The Heuristic_No_Compensate exception is raised to report that after several attempts the Compensator object has not been reached to perform the compensation while the current activity has been rolled back.

Activity_RolledBack Exception

The Activity_RolledBack exception is raised to report that the transactional activity has been rolled back.

Current interface

The **Current** interface defines operations that allow the client to manage the Activity (begin and end activities and to obtain information about the current Activity/Nested activity). Most operations provided by this **Current** interface are mainly based on those provided by the **CosTransactions::Current** interface; by this way end-users can reuse similar transactional operations which now take benefit from the Activity Service.

Since this **Current** interface aims to be layered on both OTS and the Activity Service, we assume that exceptions raised by those services are caught by this Current and re-raised to the end-user application.

How the Open Nested Transaction Current is obtained is not mandated by this specification, but could be provided using a resolve initial references("**OpenNestedTransactionCurrent**") operation on the **CORBA::ORB** interface.

The **Current** supports the following operations:

interface Current : CORBA::Current {

void activity_begin(in long timeout)
raises(CosActivity::InvalidState, CosActivity::TimeoutOutOfRange);
void activity_commit(in Compensator compensator_object,
in any compensating_data)
raises(CosActivity::NoActivity, CosTransactions::HeuristicMixed,
CosTransactions::HeuristicHazard, CosActivity::ActivityPending,
CosActivity::ChildContextPending, Activity_RolledBack,
Heuristic_Compensate, Heuristic_No_Compensate);

void activity_rollback() raises(CosActivity::NoActivity);
void activity_rollback_only() raises(CosActivity::NoActivity);

void activity_set_timeout(in long seconds) raises(CosActivity::Timeout-
OutOfRange);

Activity_Status activity_get_status();

CosActivity::ActivityToken suspend();
void resume(in CosActivity::ActivityToken)
raises(CosActivity::InvalidToken, CosActivity::InvalidParentContext);

string get_activity_name () ;
string get_transaction_name();

CosActivity::ActivityContext get_context();
CosActivity::ActivityCoordinator get_coordinator();

// Operations to access to the Transaction Service
CosTransactions::Control get_control();

// Operations to create and terminate nested transactions
void begin()
raises(CosActivity::NoActivity, CosTransactions::NoTransaction,
CosTransactions::SubtransactionsUnavailable);
void commit()
raises(CosTransactions::NotSubtransaction);
void rollback()
raises(CosTransactions::NotSubtransaction);

void rollback_only()
  raises(CosTransactions::NotSubtransaction);
};

activity_begin

A new Activity is created. If the invoking thread already has an active Activity associated with it then the newly created Activity will be nested within it. Regardless of whether or not the Activity is nested, a top-level transaction is created using the Transaction Service and associated with the newly created Activity. The invoking thread's notion of the current Activity will be changed to this Activity. If the current Activity associated with the invoking thread has completed, is completing, or has been marked as **CosActivity::CompletionStatusFailOnly**, then the INVALID_ACTIVITY exception will be thrown and the invoking threads notion of the current Activity will not be modified.

The timeout parameter is used to control the lifetime of the transactional Activity. If the Activity has not completed by the time *timeout* seconds elapses, then it is subject to being rolled back. The timeout defined by the Open Nested Current interface is not controlled by the Open Nested Service which rather relies on the underlying Activity Service to manage it. Values the timeout can have are those defined by the Activity Service.

activity_commit

The transactional Activity associated with the client thread is committed; this implicitly causes the commit of the associated transaction.

If there is no Activity associated with the calling thread, then the CosActivity::NoActivity exception will be thrown. If the Activity was begun by a thread (invoking *begin*) in the same execution environment, then the thread's Activity context is restored to its state prior to the *begin* request. Otherwise, it is set to null. If there are any encompassed active or suspended transactional, then CosActivity::ChildContextPending is raised. Only the Activity originator may call **activity_commit()**. If a call to commit 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 **activity_commit()** call is made is not the only thread on which the Activity is active, then the CosActivity::ActivityPending exception is raised.

Heuristic exceptions, CosTransactions::HeuristicMixed and CosTransactions::HeuristicHazard, raised by the underlying Transaction Service, are thrown by the Open Nested Service to the end-user.

If the *Compensator* object parameter is not null, and the Activity/transaction can commit, the Open Nested Service will register an Action with the parent activity to receive the parent outcome; failure to register the *Compensator* will cause the Activity to rollback.

If the top-level transactional activity has committed with its related transaction and compensation of a nested transactional activity has been performed the Heuristic_Compensate exception is raised.

If the transactional activity and a Compensator object responsible to compensate effect of a committed nested activity, the Heuristic_No_Compensate exception is raised.

activity_rollback

The Activity/Sub-activity associated with the client thread is rolled back. If there is no Activity associated with the calling thread then the CosActivity::NoActivity exception will be thrown. If the Activity was begun by a thread (invoking *begin*) in the same execution environment, then the thread's Activity context is restored to its state prior to the *begin* request. Otherwise, it is set to null. Any nested transactional activities are rolled back.

activity_rollback_only

If there is no Activity or Sub-activity associated with the client thread, the CosActivity::NoActivity exception is raised. Otherwise, the Activity associated with the client thread is modified so that the only possible outcome is to rollback the current Activity. Likewise, the associated transaction is also marked as rollback only.

suspend

Suspends the transactional Activity associated with the current thread with its related transaction, and any nested child scopes. A **CosActivity::ActivityToken** representing the Activity that was associated with the current thread prior to this call is returned. If the current thread is not associated with an Activity or a transaction, then nil is returned from this operation.

resume

Resumes the Activity and any nested scopes represented by the **CosActivity::ActivityToken**. The current thread becomes associated with the Activity (or transaction) represented by the token. If the **CosActivity::ActivityToken** does not represent a valid Activity (or is nil), then the CosActivity::InvalidToken exception is raised and no new association is made on the thread.

activity_set_timeout

This operation applies only to the top-level Activity. It modifies a state variable associated with the target object and affects the time-out period associated with the Activity and its associated transaction and with all nested activities created by subsequent invocations of the **activity_begin** operation. If the parameter has a nonzero value *n*, then top-level transactions created by subsequent invocations of **activity_begin** will be subject to being rolled back if they do not complete before *n* seconds after their creation. If the parameter is zero, then no application specified time-out is established.

The timeout associated with a top-level Activity is specified only at its creation and cannot be modified by subsequent Sub-activity creations. If this operation is called on a Sub-Activity the standard exception NO_PERMISSION.

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 **activity_begin**.

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.

activity_get_status

If there is no activity associated with the client thread, the StatusNoActivity value is returned. Otherwise, this operation returns the status of the activity associated with the client thread.

get_transaction_name

If there is no activity or transaction associated with the invoking thread, an empty string is returned. Otherwise, this operation returns a printable string describing the associated transaction.

get_control

If the client thread is not associated with an Activity, a null object reference is returned. Otherwise, a **CosTransactions::Control** object, created by the underlying Transaction Service, is returned that represents the transaction context currently associated with the current sub-activity. This object can be used to retrieve the transaction context associated with the current Activity.

get_context

Returns the **CosActivity::ActivityContext** of the Activity associated with the current thread. Returns null if no Activity is associated with the current thread.

get_coordinator

Returns a reference to the current Activity's **CosActvity::ActivityCoordinator**. This may be nil if no coordinator has yet been created.

begin

This operation is made available to create a nested transaction within a created transactional activity under the conditions that a previous **activity_begin** has started a transaction and that the nested transaction is supported by the underlying Transaction Service.

commit

Once invoked this operation will request the Transaction Service to commit the current nested transaction. If there is no nested transaction context associated with the current thread, the exception CosTransactions::NotSubtransaction is raised.

The client thread transaction context is modified as follows: If the nested transaction was begun by a thread (invoking *begin*) in the same execution environment, then the thread's transaction context is restored to its state prior to the begin request. Otherwise, the thread's transaction context is set to the top-level transaction context managed by the current Activity.

rollback

Once invoked this operation will request the Transaction Service to rollback the current nested transaction. If there is no nested transaction context associated with the current thread, the exception CosTransactions::NotSubtransaction is raised.

The client thread transaction context is modified as follows: If the nested transaction was begun by a thread (invoking *begin*) in the same execution environment, then the thread's transaction context is restored to its state prior to the begin request. Otherwise, the thread's transaction context is set to the top-level transaction context managed by the current Activity.

rollback_only

With this operation, the client thread is modified so that the only possible outcome is to rollback the current nested transaction. If there is no nested transaction context associated with the current thread, the exception CosTransactions::NotSubtransaction is raised.

Once invoked this operation will request the Transaction Service to mark the current sub-transaction as rollback only.

Compensator interface

The *Compensator* interface is provided to define a generic mechanism to manage the compensating action of a committed Sub-activity if one of its ancestors has rolled back.

```
interface Compensator {
void compensate(in any compensating_data);
void forget();
}
```

compensate

The *compensate* operation, defined by the application, is invoked to compensate the effects of a previously committed Sub-activity. The **compensating_data**, if not nil, may be used to perform the compensation. **compensating_data** are given at the commitment decision. The application may define a method which starts a transaction, or another Activity in order to perform the compensation.

forget

The *forget* operation is defined by the application. The application may define a method which releases the *Compensator* object.

The Open Nested Service invokes *forget* when it receives an **activity_committed** from a top-level Activity, in order to inform the *Compensator* object that the entire Activity has committed.

The Implementor's View

The Activity Service defined in the earlier part of this specification enables the development of an advanced transaction model by the definition of appropriate SignalSets, Signals and Actions for that transaction model. Therefore, in this section we describe how these entities are defined to implement the Open Nested Transaction Model requested by end-users.

Figure D-2 illustrates the relationship between an activity which may define a compensating action and the provider of the Open Nested Transaction (ONT) model.
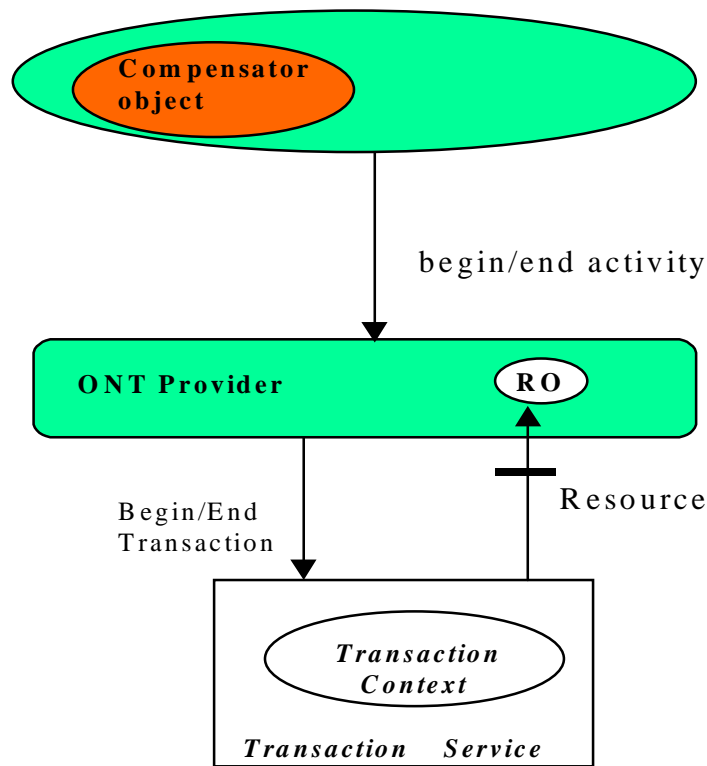

Figure D-2   Activity and ONT relationship.

Since the ONT provider is responsible for coordinating nested activities according to their final outcomes, it must have the knowledge of the outcomes of their associated transactions in the case of failure. To this aim, the ONT provider participates in the completion of the top level transaction associated with the Activity/Sub-Activity. That is,

it registers as a Resource object with the Coordinator of the transaction. Registering a Resource is a way for the ONT provider to determine if the reference of the Compensator object must logged to be retrieved in case of failure.

Once a Sub-activity has committed, its *Compensator* object must be maintained reachable in the case where either an ancestor Activity rolls back. The ONT provider and the underlying Activity Service Provider are responsible for maintaining access to *Compensator* objects. ONT providers who have registered a non nil *Compensator* object are maintained alive after their activity commitment. The path maintained to reach compensating actions is referred to as the *Compensating tree*.

The SignalSet family_outcome

Let us now describe how the Activity Service is used to coordinate a set of nested activities and how the compensating tree is maintained. In order to accomplish this we define the SignalSet *family_outcome* which contains the signals:

- *activity_rolledback*
- *activity_committed*

Defined Outcomes

The following identifiers define the Outcome structures used by the ONT protocol:

- *success_with_parent*
- *parent_does_not_exist*
- *failure_to_invoke_parent*
- *success_with_compensator*
- *failure_to_invoke_compensator*
- *heuristic_compensate_decision*
- *heuristic_cannot_compensate*

Role of the ONT provider

Upon receipt of an *activity_begin*, the Open Nested Service:

- invokes the Activity Service to create an activity
- adds a *family_outcome* SignalSet object with the created activity
- invokes OTS to create a transaction.

Upon receipt of an *activity_commit*, the Open Nested Service:

- invokes the OTS to commit the associated transaction

If the transaction commits

- if a no-null Compensator object was given in the *activity_commit*, and the commitment is related to a nested activity, the Open Nested Service adds an Action object, responsible for compensation, to the *family_outcome* SignalSet. If the adding operation fails, the Open Nested Service invokes the Compensator object to *compensate* the committed transaction using the compensate operation.

- invokes the Activity Service to complete the activity with the completion status CompletionStatusSuccess using the *family_outcome* SignalSet.

- If the Open Nested Service receives the **heuristic_compensate_decision** Outcome on the **CosActivity::Current's complete** operation, it throws for the end-user application Heuristic_Compensate exception.

- A nil outcome returned by the *complete* operation on the **CosActivity::Current** interface is interpreted as an acknowledgment to the completion with success decision.

If the transaction rolls back

- invokes the Activity Service to complete the activity with the completion status **CompletionStatusFail** using the *family_outcome* SignalSet.

- If the Open Nested Service receives the **heuristic_can_not_compensate** Outcome, it raises for the end-user application Heuristic_No_Compensate exception.

- A nil outcome returned by the *complete* operation on the **CosActivity::Current** interface is interpreted as an acknowledgment to the completion with failure decision. The Open Nested Service throws the Activity_RolledBack exception to the end-user.

Upon receipt of an **activity_rollback**, the Open Nested Service:

- invokes the OTS to rollback the associated transaction,

- invokes the Activity Service to complete the activity with the completion status **CompletionStatusFail** using the *family_outcome* SignalSet. Any enclosed transactional activity is marked to rollback.

- If the Open Nested Service receives the **heuristic_can_not_compensate** Outcome, it raises for the end-user application Heuristic_No_Compensate exception.

- A nil outcome returned by the *complete* operation on the **CosActivity::Current** interface is interpreted as an acknowledgment to the completion with failure decision.

Role of the family_outcome SignalSet

Once created and added with the **ActivityCoordinator** a family_outcome SignalSet object asks its associated **ActivityCoordinator** to obtain the reference of its parent **ActivityCoordinator** using the operation **get_parent_coordinator()**, a nil object is returned if there is no parent.

According to the activity **CompletionStatus** it receives from its associated **ActivityCoordinator** with the **set_completion_status** operation, a **family_outcome** SignalSet object provides to the **ActivityCoordinator** when request with **get_signal**, either

- the **activity_rollback** Signal (with no additional data given in the **application_specific_data** parameter), if the completion status is **CompletionStatusSuccess, or**

- the **activity_committed** Signal (with the parent **ActivityCoordinator** given in the **application_specific_data** parameter, or nil if there is no parent) if the completion status is **CompletionStatusFail.**

After providing the signal **activity_committed**, if the family SignalSet related to a nested activity receives an outcome response:

- **parent_does_not_exist** with **set_response** indicating that an Action fails to be registered with the parent **ActivityCoordinator** because it does not exist, the SignalSet indicates to the **ActivityCoordinator** that a subsequent signal shall be sent to that Action. This next signal is **activity_rollback**.

- **failure_to_invoke_parent** with **set_response** indicating that an Action fails to be registered with the parent **ActivityCoordinator** due to a transient failure or a communication failure, the SignalSet indicates to the **ActivityCoordinator** that the same signal, **activity_committed** shall be sent to that Action. However, if the SignalSet receives the same Outcome **failure_to_invoke_parent** several times it can decide to issue the **activity_rolledback** signal.

After providing the Signal **activity_committed**, if the **family_outcome** SignalSet related to the top-level activity receives

- the outcome response **failure_to_invoke_compensator** with **set_response** indicating that an Action fails to invoke forget on the Compensator object due to a transient failure or a communication failure, the SignalSet indicates to the **ActivityCoordinator** that the same signal, **activity_committed** shall be sent to that Action. family_outcome.

- an Outcome indicating that the top-level **ActivityCoordinator** fails to invoke an Action with the signal **activity_commited** because it does not longer exist, it informs the SignalSet. Once requested to obtain the final outcome with **get_outcome**, the **family_outcome** SignalSet will return the outcome **heuristic_compensate_decision**.

After providing the signal **activity_rolledback**, if a **family_outcome** SignalSet receives

- the Outcome response **failure_to_invoke_compensator** with **set_response** indicating that an Action fails to invoke *compensate* on the Compensator object due to a transient failure or a communication failure, the SignalSet indicates to the **ActivityCoordinator** that the same signal, **activity_rolledback** shall be sent to that Action. However, after several retrying, the SignalSet can decide to abandon the compensation. Once requested to obtain the final outcome with **get_outcome**, the **family_outcome** SignalSet will return the outcome **heuristic_can_not_compensate**.

- an Outcome indicating that the **ActivityCoordinator** fails to invoke an Action with the signal **activity_rolledback** due a communication failure exception, it informs the SignalSet. The family_outcome SignalSet ignores that Action, there is no additional signal or retrying. After a timeout, or once restarted, the ONT provider can ask the **ActivityCoordinator** to obtain the status of the activity; if the **OBJECT_DOES_NOT_EXIST** is returned indicating that the activity does not exist, the ONT provider will presume that it has rolled back and it invokes compensate on the Compensator object.

Role of the Action registered with the family_outcome SignalSet

If an Action registered to the **family_outcome** signalSet receives the **activity_committed** signal with a non-nil parent **ActivityCoordinator**, it registers with that parent so that it can be informed about its outcome. The Action has knowledge that it represents an Activity that has, nominally, completed successfully, but which may need to be compensated later. The Action then returns the Outcome **success_with_parent**. If the registration with the parent fails because it does not exist, it return the Outcome **parent_does_not_exist**. If the registration with the parent fails due to a transient or communication failure, it return the Outcome **failure_to_invoke_parent**.

This recursively ends up in having all Compensator objects listed in the "family Actions" and having the family Actions registered to the top-level **family_outcome** signalSet as illustrated in Figure D-3.
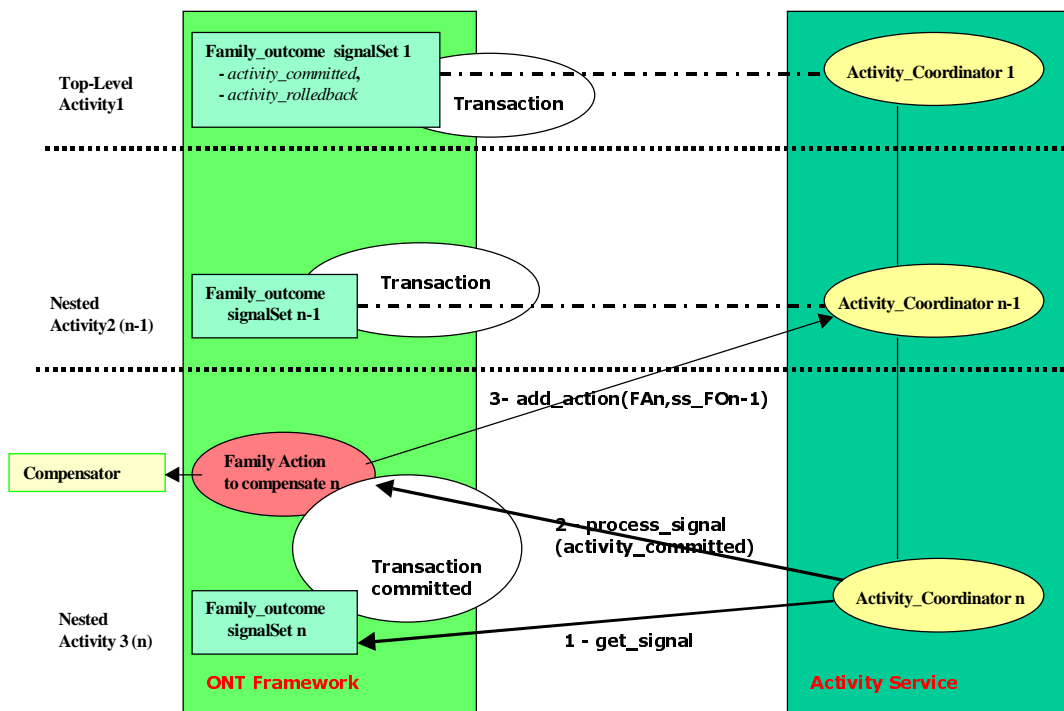

Figure D-3 Transactional Activity commitment and Compensation registrations

If an Action registered to the **family_outcome** signalSet receives the **activity_committed** signal with a nil parent **ActivityCoordinator**, it invokes a forget operation on the Compensator object to inform it about the final completion. If the Action fails to invoke the Compensator object it returns the outcome **failure_to_invoke_compensator**.

If an Action registered to the **family_outcome** signalSet receives the **activity_rollback** signal, it invokes the *compensate* operation on the Compensator object as described in Figure D-4. If the Action fails to invoke the Compensator object it returns the outcome **failure_to_invoke_compensator**.
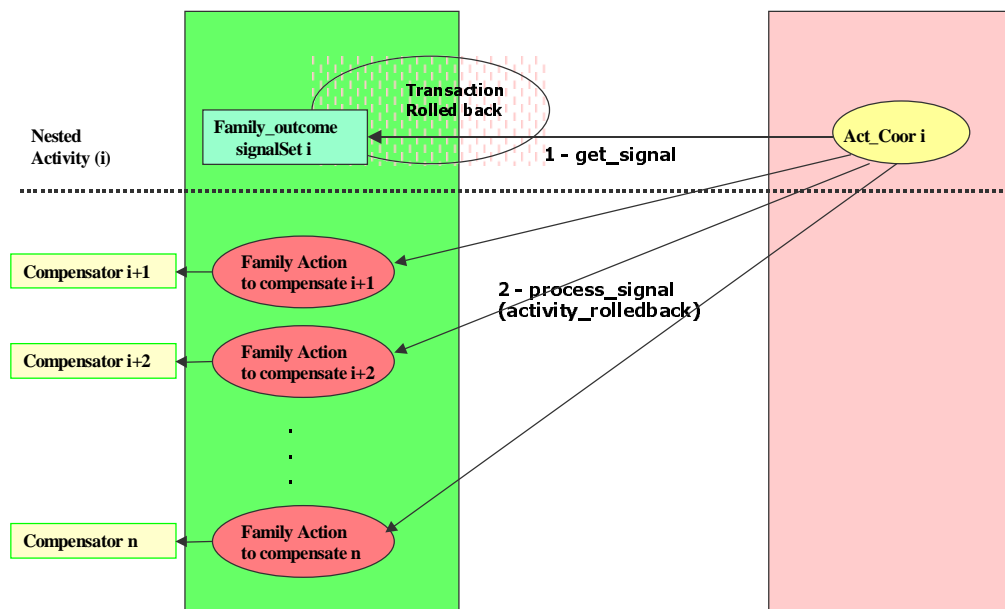
Figure D-4   Compensation on Activity rollback

After a timeout or once restarted, the Open Nested Service has the responsibility to inquiry of its associated activity using the **get_status** operation on the **CosActivity::ActivityCoordinator**. If the **ActivityCoordinator** no longer exists, it invokes *compensate* on the Compensator object. It has the responsibility to retry the *compensate* method in case of failure.

An application programmer may invoke the creation of a transaction or an activity within the *compensate* operation. However if that transaction rolls back or the activity completes with failure, it is up to application to retry. The Open Nested Service which invokes *compensate* is not responsible for its behavior, but only responsible to reach the Compensator object.

Open Nested Transaction IDL

#ifndef OPEN_NESTED_IDL_

```
#define OPEN_NESTED_IDL_

#include <orb.idl>
#include <CosTransactions.idl>
#include <CosActivity.idl>
module OpenNested
{
  enum Activity_Status {
StatusActive,
StatusNoActivity,
StatusMarkedRollback,
StatusRollingBack,
StatusCommitting,
StatusRolledBack,
StatusCommitted,
StatusToCompensate,
StatusUnknown
  };

  exception Heuristic_Compensate {};
  exception Heuristic_No_Compensate {};
  exception Activity_RolledBack {};

  interface Compensator;

  interface Current : CORBA::Current {

    void activity_begin(in long timeout)
       raises(CosActivity::InvalidState, CosActivity::TimeoutOutOfRange);
    void activity_commit(in Compensator compensator_object, in any
compensate_data)
        raises(CosActivity::NoActivity, CosTransactions::HeuristicMixed,
            CosTransactions::HeuristicHazard, CosActivity::ActivityPend-
ing,
           CosActivity::ChildContextPending, Activity_RolleBack,
      Heuristic_Compensate, Heuristic_No_Compensate);

    void activity_rollback() raises(CosActivity::NoActivity);
    void activity_rollback_only() raises(CosActivity::NoActivity);

    void activity_set_timeout(in long seconds) raises(CosActivity::Timeout-
OutOfRange);
    Activity_Status activity_get_status();

    CosActivity::ActivityToken suspend();
    void resume(CosActivity::ActivityToken)
   raises(CosActivity::InvalidToken, CosActivity::InvalidParentContext);

    string get_activity_name ();
    string get_transaction_name();
```

```
        CosActivity::ActivityContext get_context();
        CosActivity::ActivityCoordinator get_coordinator();

        // Operations to access to theTransaction Service
        CosTransactions::Control get_control();

        // Operations to create and terminate  nested transactions
        void begin() raises(CosActivity::NoActivity, CosTransactions::NoTrans-
action,
                CosTransactions::SubtransactionsUnavailable);
        void commit() raises(CosTransactions::NotSubtransaction);
        void rollback() raises(CosTransactions::NotSubtransaction);
        void rollback_only() raises(CosTransactions::NotSubtransaction);
    };

    interface Compensator {
void compensate(in any compensating_data);
void forget();
    };
};
#endif
```