A framework for implementing extended transactions

M. C. Little†, S. K. Shrivastava‡ and S. M. Wheater†
†HP Arjuna Labs, Newcastle upon Tyne, England,
‡Computing Science Department, University of Newcastle, Newcastle upon Tyne, England

1. Introduction

Structuring certain applications from long-running transactions can reduce the amount of concurrency within an application or in the event of failures require work to be performed again, and hence adversely affect application performance. For example, 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; in the event of the transaction rolling back, however, certain compensation activities may be necessary to restore the system to a consistent state. Such compensation activities, which may perform forward or backward recovery, will typically be application specific, may not be necessary at all, or may be more efficiently dealt with by the application.

There are a number of different extensions to the standard transaction model that have been proposed to address specific application needs, that may not be easily or efficiently addressed through the use of traditional transactions:

- *Nested transactions*: permits a finer control over recovery and concurrency [1]. The outermost transaction of such a hierarchy is referred to as the top-level transaction. The permanence of effect property is only possessed by the top-level transaction, whereas the commits of nested transactions (subtransactions) are provisional upon the commit/abort of an enclosing transaction.
- *Type specific concurrency control*: concurrent read/write or write/write operations are permitted on an object from different transactions provided these operations can be shown to be non-interfering [2].
- *Independent top-level transactions*: with this model it is possible to invoke a top-level transaction from within another (possibly deeply nested) transaction [3]. If the logically enclosing transaction rolls back, this does not lead to the rollback of the independent top-level transaction, which can commit or rollback independently. In the event that the enclosing transaction rolls back, compensation may be required, but this is typically left up to the application.
- *Structured top-level transactions*: long-running top-level transactions can be structured as many independent, short-duration top-level transactions [4]. This allows an activity to acquire and use resources for only the required duration. In the event of failures, to obtain transactional semantics for the entire duration may require compensations for forward or backward recovery.

What this range of extended transaction models illustrate is that a single model is not sufficient for all applications. Therefore, is it possible to develop a framework within which all of these models can be supported, and also facilitate the development of other models? This was the question asked by the Object Management Group when it began its work on attempting to standardise extended transaction models. In this paper

we shall given an overview of the results of the work we performed with IBM, Iona and others in producing the final *Activity Service* OMG specification that attempts to answer that question [5].

2. The activity framework

The framework to be outlined provides a low-level infrastructure to support the coordination and control of abstract, application specific entities. These entities (*activities*) may use ACID transactions, they may use weaker forms of serializability, or they may not be transactional at all; the framework is only concerned with their control and co-ordination, leaving the semantics of such activities to the application programmer. If the activities use transactions, then the famework implementation will ensure that transaction contexts are managed correctly, e.g., contexts flow across execution environments and transactions that are begun within the scope of an activity are terminated before the activity terminates. A very high level view of the role of the Activity Service is shown in Figure 1.

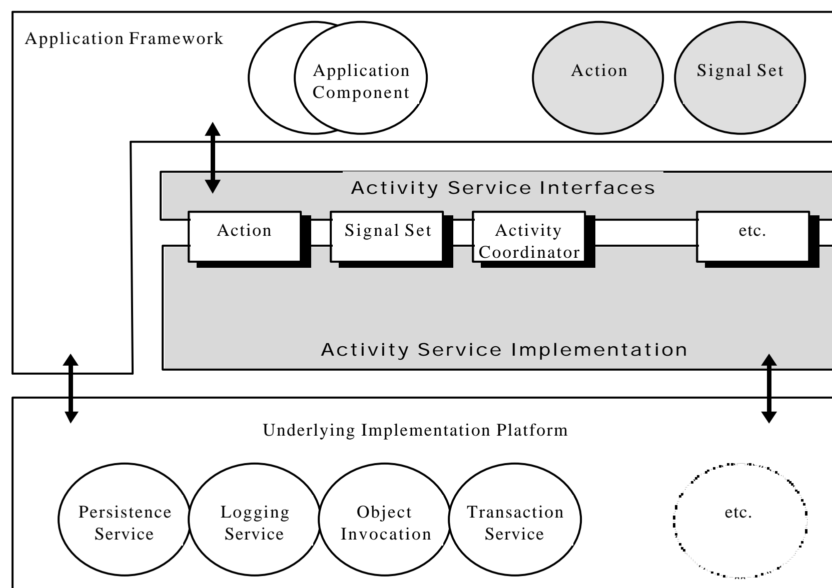

Figure 1: The role of the Activity Service.

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. 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 messages are communicated to entities (*Actions*) registered with an activity through *Signals*. Signals can be used to infer a flow of control during the execution of an application. For example, the termination of one activity may initiate the start/restart of other activities in a workflow-like environment.

An activity may run for an arbitrary length of time, and may use ACID transactions at any point during its lifetime. It was extremely important from an industrial point of view that this framework could work with existing transactional systems. These ACID

transactions may be provided by an OTS-compliant transaction service implementation. For example, consider Figure 2, 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. The Activity Service is responsible for distributing both the activity and transaction contexts between execution environments in order that the hierarchy can be fully distributed.

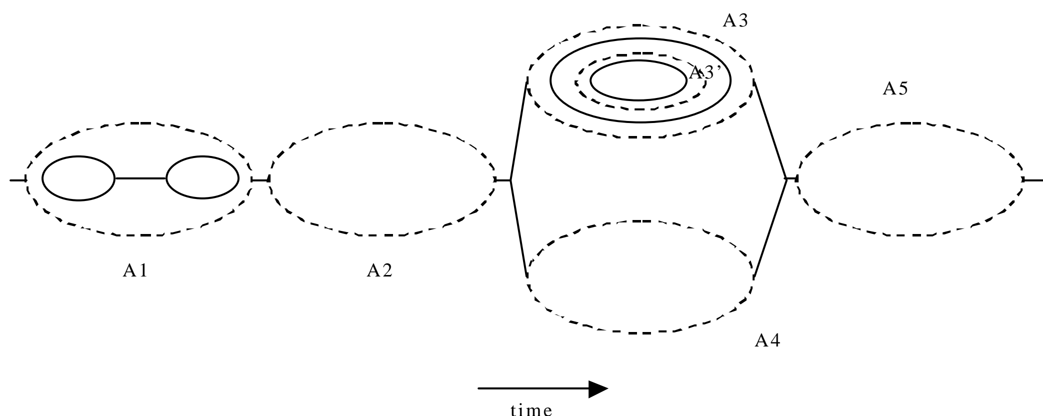

Figure 2: Activity and transaction relationship.

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*. Importantly for application consistency, tt is not possible to suspend an activity without suspending all of its enclosed transactions. In addition, suspending a transaction which has enclosed activities will also suspend those activities.

2.1 Activity coordination and control

An activity may decide to transmit activity specific data (Signals) to any number of other activities at specific times during its lifetime, e.g., when it terminates. The information encoded within a Signal can be arbitrary, and will depend upon the implementation of the extended transaction model. To drive the activity interactions an *activity coordinator* is associated with each activity. Activities that require to be informed when another activity sends a specific Signal can register with that activity's coordinator. The coordinator's role is to send Signals to all registered participants and to deal with the outcomes generated.

Importantly, the implementation of the coordinator will depend upon the type of extended transaction model being used. For example, if a Sagas type model is in use [6] then a compensation Signal may be required to be sent to activities if a failure has happened, whereas a coordinator for a strict transactional model may require to send a

Signal informing participants to rollback. One of the keys to the extensibility of this framework is the *Signal Set* whose implemented behaviour is peculiar to the kind of extended transaction. The Signal Set is essentially a Signal factory that produces Signals that are sent to Actions and processes the results: it is the coordination logic. Similarly, the behaviour of an Action will be peculiar to the extended transaction model of which it is a part.

Therefore, to enable the coordinator to be configurable for different transaction models, the coordinator delegates all Signal control to a Signal Set. The intelligence about which Signal to send to an activity is hidden within a Signal Set and may be as complex or as simple as is required. The coordinator itself is therefore extremely simple and generic, i.e., a single coordinator implementation can be used for all extended transaction implementations.

As new types of extended transaction emerge, new Signal Set instances and associated Actions and Signals will be created. This allows a single implementation of this framework to serve a large variety of extended transaction models, each with its own action and Signal Set implementations. The *framework implementation* will not need to know the behaviour which is encapsulated in the actions and Signal Sets it is given, merely interacting with their opaque interfaces in an entirely uniform and transparent way.

2.2 Composite activities

An activity which contains component activities, may impose a requirement on the Activity Service implementation for managing these component activities. It may be necessary to determine whether these component activities worked as specified or failed and how to map their (non-) completion to the enclosing activity's outcome. This is true whether the activities are strictly parallel, strictly sequential or a complex structure. In general, an activity that needs to co-ordinate the outcomes of component activities has to know what state each component activity is in, i.e., which are active, which have completed and what their outcomes were, and which activities failed to complete.

Another activity may therefore be required to handle the sub-activity outcomes so that control flows can be made explicit. This activity determines the collective outcome of the component activities in the light of the various component failure and success situations. The activity framework does not specify how the activities should be coordinated, only providing interfaces for coordination to occur. The coordination may therefore be performed in a manner most suitable to the application or extended transaction model. For example, a scripting language may be required to assist the application programmer in a workflow-like manner [7].

2.3 Activity failures

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

- if the activity was involved within a transaction, then any state changes it may have been making when the failure occurred will eventually be recovered automatically by the transaction service.
- if the activity was not involved within a transaction, then application specific compensation may be required.
- an application that consisted of the (possibly parallel) execution of many activities (transactional or not) may still require some form of compensation to "recover" committed state changes made by prior activities.

Rather than distinguish between compensating and non-compensating activities, we consider that the compensation of the state changes made by an activity is simply the role of another activity. A compensating activity is simply performing further work on behalf of the application. Just as application programmers are expected to write "normal" activities, they will therefore also be required to write "compensating" activities, if such are needed. In general, it is only application programmers who possess sufficient information about the role of data within the application and how it has been manipulated over time to be able to compensate for the failure of activities.

3. Using the framework

Although most effort has been concentrated on the development of the generic framework, there has been initial work on using it to implement specific extended transaction models. This includes a Sagas-like model [6], where coordination of top-level transactions occurs, and an implementation of Open Nested Transactions [5]. Preliminary use of this framework has shown that it can support both of these extended models successfully. In order to determine whether the framework can support other models we intend to use it in a number of environments that require different types of extended transaction implementations. If deficiencies with the framework appear then we shall feed them back into the original OMG specification.

4. References

[1] J. E. B. Moss, "Nested Transactions: an approach to reliable distributed computing", Ph.D. Thesis 260, MIT, Cambridge, MA, April 1981.

[2] P. M. Shwarz and A. Z. Spector, "Synchronizing Shared Abstract Types", ACM Transactions on Computer Systems, Vol. 2, No. 3, August 1984, pp. 223-250.

[3] B. Liskov and R. Scheifler, "Guardians and actions: linguistic support for robust distributed programs", ACM TOPLAS, Vol. 5, No. 3, July 1983, pp. 381-404.

[4] Nortel, supported by the University of Newcastle upon Tyne, "OMG document bom/98-03-01", submission for the OMG Business Object Domain Task Force (BODTF): Workflow Management Facility, 1998.

[5] OMG, *Additional Structuring Mechanisms for the OTS Specification*, September 2000, document orbos/2000-04-02.

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

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