Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (0223) 323010**
INTERNATIONAL: **+44 223 323010**
FAX: **+44 223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

Extended Transaction Framework: Technical Overview

John Warne

Abstract

This document presents a technical overview of the proposed ANSA Phase III Extended Transaction Framework (ETF): a set of principles, concepts, models, and tools for the construction and federation of transactional-based, business application processes operating in large-scale, heterogeneous, open distributed systems.

APM.1060.00.03 **Draft** 10 May 1994

Request for Comments (confidential to ANSA consortium for 2 years)

Distribution:
Supersedes:
Superseded by:

Extended Transaction Framework: Technical Overview

Extended Transaction Framework: Technical Overview

John Warne

APM.1060.00.03

10 May 1994

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by Architecture Projects Management Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

Architecture Projects Management Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(0223) 323010
INTERNATIONAL	+44 223 323010
FAX	+44 223 359779
E-MAIL	apm@ansa.co.uk

Contents

1 Introduction

This document presents a technical overview of the proposed ANSA Phase III Extended Transaction Framework (ETF): a set of principles, concepts, models, and tools for the construction and federation of transactional-based, business application processes operating in large-scale, heterogeneous, open distributed systems.

It is a primary goal that the ETF shall be of sufficient generality to support a wide range of telecommunications and other business applications. Moreover, it is expected that these business applications may vary widely in customer service functionality and Quality of Service (QoS) requirements. Such tailoring will almost certainly result in diversity of distributed transaction processing and data management activities. Inevitably, different classes of customer specialisation will necessitate the use of different transaction models with differing concurrency control methods, recovery procedures, resource placement, migration and replication strategies, and timeliness (execution responsiveness) guarantees.

The objective of the ETF is to enable such application diversity to interoperate effectively.

1.1 Motivation

The benefits of using traditional ACID transactions as dependable computations for preserving the integrity and consistency of long-lived (possibly distributed) information are well known and are described in detail elsewhere (e.g., [BERNSTEIN 87] and [GRAY 93]).

Although the ETF described in this document builds on the earlier work of traditional transactions, it also extends that work by introducing concepts and techniques which enable the structure and behaviour of transactions to be tailored more flexibly to the requirements of a wider range of application domains than those domains which simply encapsulate traditional (transaction) databases applications.

The motivation for the ETF is best explained by reviewing the properties and benefits of the traditional transaction model, and then by revealing the limitations of the model and its required extensions.

1.1.1 Review of the traditional transaction model

The semantics of the traditional transaction are constrained by the fundamental properties of atomicity, consistency, isolation, and durability (collectively known as the ACID properties [BERNSTEIN 87]).

With these properties in place, each transaction is guaranteed to be

- **failure atomic**: it is *all-or-nothing* in that its computational results are durably recorded in system state if it commits; or its results are discarded if it aborts
- **serializable**: its computational results produced in a schedule of concurrent transactions can always be reproduced in some serial schedule of the same transactions.

These two behavioural characteristics are used as benchmarks for measuring the correct execution of concurrent (traditional) transactions. Failure atomicity safeguards correct transactions from incorrectly manipulating data in the event of system failures; and serializability ensures conflict-free access to data by concurrent transactions.

1.1.2 Benefits of the traditional transaction model

There can be no doubt about the benefits derived from applying the traditional transaction model:

1. its application is most effective for controlling short, concurrent computations in competitive, shared data environments;
2. its execution properties immensely simplify the task of programming applications requiring dependable data management support;
3. its underlying techniques and algorithms for implementing concurrency control, atomic commitment, and failure recovery have proven to be acceptably efficient for commercial practice.

There can also be no doubt that the use of this model will continue in those application domains where the above benefits are not only vital, but also sufficient. Accordingly, the proposed ETF must provide architectural support for traditional transactions.

1.1.3 Limitations of the traditional transaction model

Although highly effective in competitive environments that support, say, conventional database applications, the traditional transaction model, with its strict ACID properties, is frequently found lacking in functionality, flexibility, and performance when used for applications that involve collaborative, and/or long-lived activities. Such applications typically require multiple transactions to share resources (distributed object state) in complex ways, for example, by exchanging access to resources without any one of them necessarily having to terminate (commit or abort) its execution. Moreover, the execution periods for these applications may continue for hours, days, months, or even longer!

The strict isolation property of the traditional transaction model clearly precludes the possibility of inter-transaction cooperation!

1.1.4 Extended transaction models

In response to the inflexibility of the traditional transaction model, several new ‘extended transaction models’ have emerged, each defining a specific type of transaction, for example, Cooperative CAD Transactions [NODINE 92], Cooperative SEE Transactions [HEILER 92], Split Transactions [PU 88], and Coloured Transactions [WHEATER 90].

(A comprehensive description of several extended transaction models can be found in [ELMAGARMID 92].)

While these extended models differ in various forms, they all share a common feature, namely, the ability to relax the isolation property of ACID and consequently control the degree with which one transaction can cooperate with others in the use of shared resources.

The correctness criterion of each model is typically application-specific and is defined by the allowed behaviour of its active transaction components and the interactions between them. Such behaviour is constrained by the execution dependencies that a transaction can develop with respect to other transactions. These dependencies impose controls over the circumstances in which cooperative transactions may legitimately commit or abort and thereby prevent the occurrence of “unacceptable” (inconsistent) results.

It is observed in [CHRYSANTHIS 90] that irrespective of how successful these extended transaction models are in supporting their intended application domains, they merely represent points within the spectrum of interactions possible within competitive and cooperative environments. Therefore, they each capture only a subset of the interactions to be found in any complex information system.

The proposed ETF is intended to address these concerns.

1.2 Comments on the proposed extended transaction framework

As an architectural framework, the ETF is not simply another extended transaction model, but rather a unifying vehicle for the specification, construction, and inter-operation of different types of transaction models.

1.3 Assumptions

It is assumed that readers of this document are familiar with the basic principles and techniques of atomic transactions as defined in any authoritative textbook on the subject (e.g. [BERNSTEIN 87], [GRAY 93]). Some additional familiarity with the nested transaction model [MOSS 81] would also be helpful, particularly in the context of an object-based environment [WARNE 93].

1.4 Document organisation

The remainder of this document is divided into three chapters. Chapter 2 briefly reviews the research results upon which several of the principles, concepts, and structures of the ETF are based. Chapter 3 presents the architectural vision of the ETF and provides a high level view of the processes and components that are needed to support the construction of extended transactions and their underlying run-time environments. Chapter 4 reviews the work presented in this document and comments on future work.

2 Building on the successful results of others

Many of the principles and concepts of the proposed ETF are based on the successful results of several recent research initiatives that have focused on modelling techniques for thinking about or realizing extended transaction models. A summary of these initiatives are given below.

2.1 ACTA meta-model

ACTA [CHRYSANTHIS 92, RAMAMRITHAM 92] is a formal framework designed to specify, analyse and synthesize transaction models using a number of conceptual building blocks for expressing dependencies between transaction components and for expressing the rules by which these dependent transactions can delegate (share) access to and responsibility for object resources in a controlled manner.

The ACTA model has been used successfully to contrast the semantics of several existing transaction models, including traditional transactions [BERNSTEIN 87], nested transactions [MOSS 81], split transactions [PU 88], and sagas [GARCIA-MOLINA 87]. It has also been used for the methodical development of new transaction models, for example, the nested-split transaction model which combines aspects of the nested and split transaction models.

The ETF identifies a number of new architectural primitives for controlling the behaviour of extended transaction models. The inspiration for these primitives stems from the abstract concepts of the ACTA meta-model.

2.2 Transaction-based work flow models

Transactional-based work flow models [DAYAL 90, SHETH 93, RUSINKIEWICZ 93, GEORGAKOPOULOS 92] are designed to enable collections of different extended transactions to participate as subtasks of transactional-based, application-specific business processes.

With individual extended transactions models, the flow of control between constituent transactions is traditionally encoded into the application-specific transactions themselves.

This approach contrasts sharply with advanced work flow models which prescribe workflow facilities as separate layers on top of application-level interfaces. In this way, the workflow scheduler interacts with each application and orchestrates the sequencing and communication between the applications as a whole. Moreover, the workflow scheduler is itself executed as an extended transaction with all the concomitant support of controlled execution dependability.

Workflow models typically make use of rules to enforce correctness of dependencies and flows between workflow components. These rules define the

conditions under which transaction dependencies can be resolved and their consequent actions.

The ETF is specifically intended to provide an environment in which application-independent, extended transaction models can be combined with application objects to generate application-specific, multi-transaction, business workflows.

3 Architectural Vision

The main principle underpinning the architectural structure of the ETF is that all extended transaction models are each formed by a collection of flat transactions with a set of execution dependencies between members of the collection and a set of rules which determine how members can share access to acquired object resources.

Essentially, the set of transaction dependencies and rules for resource sharing for any specific extended transaction model conjointly determine the behavioural characteristics of the model's transaction type(s) in terms of four basic attributes:

- **visibility**, the degree with which members of the extended transaction are able to observe each others effects before the transaction as a whole terminates its execution and commits or aborts;
- **correctness**, the acceptable effects on system state that members are permitted to produce;
- **permanence**, the rules by which the effects of members are allowed to be recorded in the stable state of the system;
- **recoverability**, the capability of an extended transaction as a whole, or its members in part, in the event of failure, to recover and take the system to some state that is considered correct.

Different degrees of these attributes can be realized, firstly, by relaxing the isolation property of member transactions in order that they can observe each others computational effects; and, secondly, by enforcing desired execution and system state access behaviours through appropriate transaction dependency and object resource delegation controls.

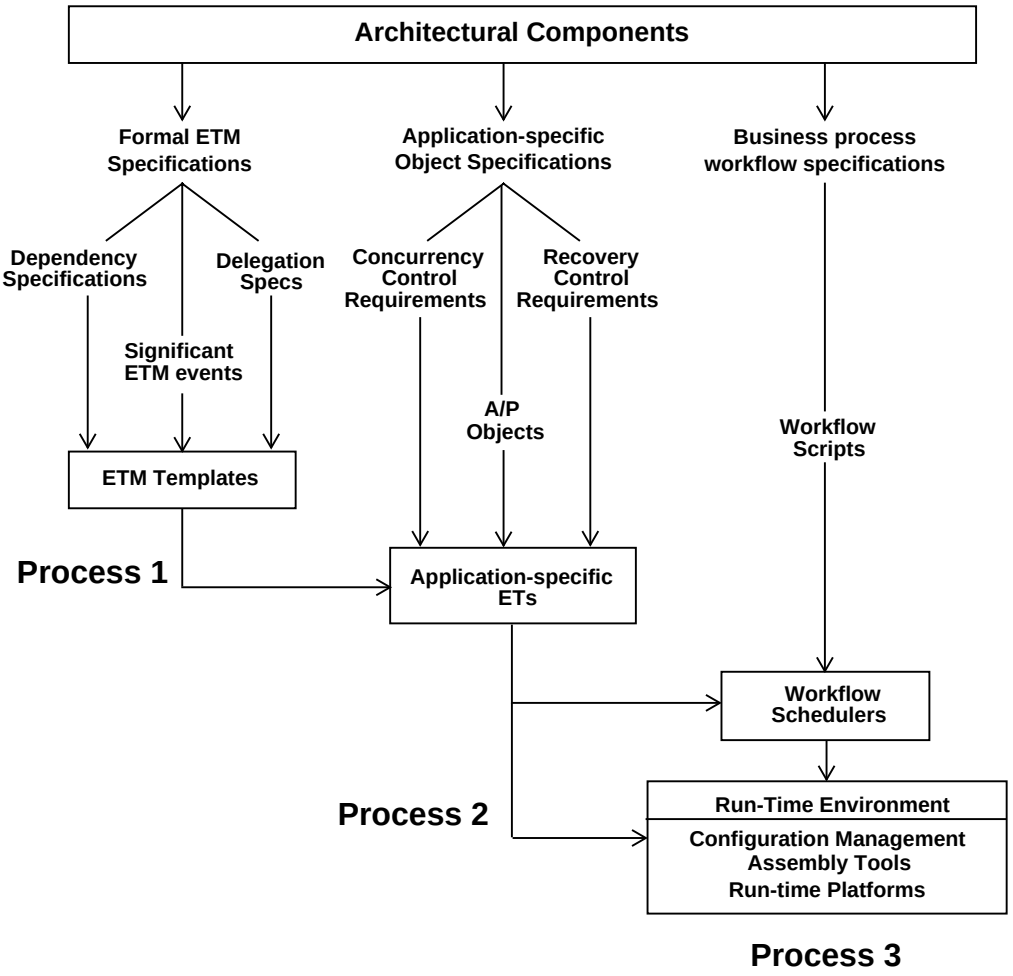
The remainder of this chapter elaborates these aspects. It presents the component structure of the proposed ETF with its underlying processes and mechanisms. It outlines an architecture for the construction of arbitrarily complex extended transaction models and their requisite run-time support facilities.

3.1 Overall structure of the ETF

The combined process and structural composition of the ETF is illustrated in Figure 3.1.From an operational perspective, the ETF comprises three related processes:

- **Process 1** constructs Extended Transaction Model (ETM) templates using elements derived from the formal specifications of the models.
- **Process 2** constructs application-specific extended transactions (ETs) by combining ETM templates and application (A/P) object specifications, including any specific application requirements for concurrency control and recovery.

Figure 3.1: Composition of the ETF



- **Process 3** constructs workflow schedulers for controlling the execution of business processes defined by workflow scripts describing the run-time flow and configuration of application-specific ETs.

The components of each of these processes are described in the following.

3.2 ETM Templates

Each ETM template describes the components of a model in terms of three run-time elements:

1. the dependencies between member transaction (e.g., transaction T_i can commit if transaction T_j commits);
2. the delegation rules for specifying how member transactions may pass access to shared resources among themselves;
3. the transaction members and the transaction management operations used to control the run-time behaviour of the extended transaction.

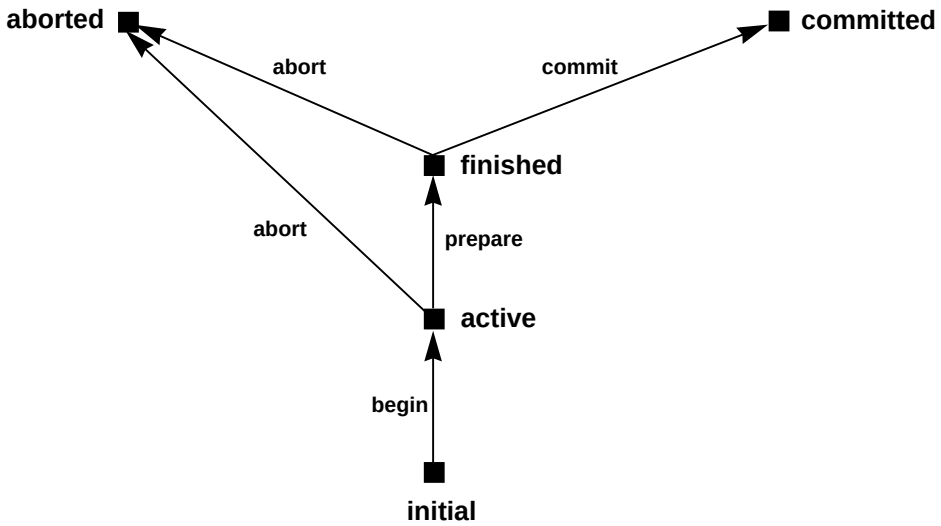
In this form, the template has not yet been bound to any application objects. It simply represents the generic run-time infrastructure for a particular class of extended transactions (e.g. nested transactions).

The abstract specifications for the elements listed above are obtained from the formal specification of the extended transaction model. In this overview, the formal specification language is based on an extension of the notation defined in the ACTA meta-model [CHRYSANTHIS 92].

3.3 Transaction state transitions, operation events and dependencies

The state transitions of a single transaction and the operational events that cause them are shown in Figure 3.2.

Figure 3.2: Transaction state transition diagram



From its `initial` state, a transaction, via a `begin` operation, assumes its `active` state. Eventually, the transaction will either `abort` and move to the `aborted` state, or move to the `finished` state by issuing a `prepare` operation. Finally, from the `finished` state, the transaction can either `commit` (i.e., make the `committed` transition), or `abort` (i.e., make the `aborted` transition). The two operational events `commit` and `abort` are said to move the transaction to a `terminated` state. The `commit` and `abort` events are correspondingly called **transaction termination events**.

Transaction dependencies specify the rules by which transaction members of an extended transaction model can affect each others state transitions and thus control each others execution.

3.3.1 Transaction dependency specifications

A transaction dependency specification is a set of inter-transaction dependencies, each of which has the generic form

$$T_j \text{ dependency: } x \ T_i$$

where T_j and T_i are transaction members of a given extended transaction model in which they are related by a dependency of type `x`.

In its simplest form, a dependency type is expressed in terms of the occurrence of an event pair (eT_j, eT_i) pertaining to a transaction pair (T_j,T_i) in a conceptual execution history `H`.

In general, an event is any transactional happening of interest that is permitted to be recorded in history H. Each event is typically a request to invoke a transaction management operation.

3.3.1.1 Dependency forms

There are two basic forms of transaction dependencies, the first of which (*form 1*) is

$$(eT_j \in H) \Rightarrow ((eT_i \in H) \wedge (eT_i < eT_j))$$

which has the meaning "event eT_j can be recorded in the history H if event eT_i is recorded in H and thus eT_i precedes eT_j in the recording process".[1]

Another way of saying this is that the transaction management operation for event eT_j can be executed only if the corresponding operation for event eT_i has already been executed.

It is to be emphasised that such dependencies do not assert that eT_j will necessarily occur, even if eT_i occurs and is recorded in H.

In general, dependencies of *form 1* can be expressed simply as

$$(eT_j \in H) \Rightarrow (eT_i \in H)$$

where the ordering $(eT_i < eT_j)$ is understood.

The second basic form of transaction dependency (*form 2*) is

$$(eT_j \in H) \Leftrightarrow (eT_i \in H)$$

which has the stronger meaning "if eT_j occurs then eT_i occurs".[2]

In this case, no specific ordering is implied. Unlike *form 1*, where eT_j may not necessarily occur, in this case, if eT_i is recorded in H then eT_j is also recorded in H, and vice versa.

3.3.1.2 Alternative events

It is often necessary to express the requirement that a specific event be recorded in the history H should it be determined (or assumed) that some other dependent event will never occur.

For *form 1* (Section 3.3.1.1), the required expression is

$$((eT_j \in H) \Rightarrow (eT_i \in H))::(e'T_j \in H)$$

with the meaning "if event eT_j occurs and it is determined (and thus consequently expected) that event eT_i will never occur, then the alternative event $e'T_j$ is to be recorded in the history.

For *form 2* (Section 3.3.1.1), the required expression is

$$((eT_j \in H) \Leftrightarrow (eT_i \in H))::(e'T_j, e'T_i \in H)$$

with the alternative events $e'T_j$ and $e'T_i$.

(Note that alternative events are triggered by the ETF infrastructure on behalf of the transaction(s) under consideration.)

---

1. In *form1* expressions, the symbol $\Rightarrow$ has its usual meaning of implication and the symbol < means that events appearing on the left hand side are recorded in the history H before events on the right hand side.

2. In *form 2* expressions, the symbol $\Leftrightarrow$ means "either both or neither".

3.3.2 Examples of transaction dependencies

Some practical examples of transaction dependencies follow in which the events of interest are invocations of the transaction management operations `begin`, `commit` and `abort`. However, these examples are by no means exhaustive.The proposed ETF is a general framework intended to be unrestricted with respect to defining required dependency types based on any significant events (operation invocations). Such generality, however, is the subject of a subsequent document.

3.3.2.1 Example 1: commit dependency

Transaction T_j is commit dependent on transaction T_i:

$$((commitT_j \in H) \Rightarrow (commitT_i \in H))::(abortT_j \in H)$$

T_j can commit only if T_i commits, else T_j must abort. However, T_j can always abort no matter what T_i does.

A shorthand notation for this type of dependency is (T_j `CD` T_i).

3.3.2.2 Example 2: strong commit dependency

Transaction T_j has a strong commit dependency on transaction T_i:

$$((commitT_j \in H) \Leftrightarrow (commitT_i \in H))::(abortT_j, abortT_i \in H)$$

Both T_j and T_i commit, or both abort

A shorthand notation for this type of dependency is (T_j `SCD` T_i).

3.3.2.3 Example 3: commit-on-abort dependency

Transaction T_j has a commit-on-abort dependency with respect to transaction T_i:

$$((commitT_j \in H) \Rightarrow (abortT_i \in H))::(abortT_j \in H)$$

T_j can commit only if T_i aborts, else T_j must abort. However, T_j can abort no matter what T_i does.

A shorthand notation for this type of dependency is (T_j `CAB` T_i).

3.3.2.4 Example 4: begin-on-commit dependency

Transaction T_j has a begin-on-commit dependency with respect to transaction T_i:

$$(beginT_j \in H) \Rightarrow (commitT_i \in H)$$

T_j can begin only if T_i commits. In this case, no alternative cases are required, since the condition is absolute.

A shorthand notation for this type of dependency is (T_j `BOC` T_i).

3.3.3 Composite dependencies

Simple dependency expressions can be extended to express composite (multiple) dependencies between one transaction and several other transactions.

For *form 1* (Section 3.3.1.1) the composite dependency

$$(eT_x \in H) \Rightarrow ((eT_i \in H) \wedge (eT_j \in H) \wedge (eT_k \in H) \wedge \ldots\ldots\ldots))$$

means that the event eT_x for transaction x can be recorded in history H only if all events eT_i, eT_j, eT_k, ... specified in the right-hand side of the expression are recorded in H.

For example, the dependency

$$(commitT_x \in H) \Rightarrow ((commitT_i \in H) \wedge (commitT_j \in H) \wedge (abortT_k \in H))$$

allows T_x to commit, but only if T_i and T_j both commit and T_k aborts.

For *form 2* (Section 3.3.1.1), the following composite dependency

$$(commitT_x \in H) \Leftrightarrow ((commitT_i \in H) \wedge (commitT_j \in H) \wedge (commitT_k \in H))$$

means "if T_x commits then T_i, T_j and T_k commit".

This example epitomises the required commit outcome of an atomic commit agreement protocol.

Composite dependency specifications may also include statements about alternative events (Section 3.3.1.2). For example, the immediately preceding composite dependency would specify an alternative abort action for each of the transactions specified.

3.3.4 Run-time structures for dependencies

Each dependency needs a run-time mechanism that enables it to be evaluated and actioned. An obvious implementation is for one (or all) of the dependent transactions to interrogate the status of the other(s) and thereafter instigate appropriate action based on the response(s).

Consider dependency expressions of the form

$$((eT_j \in H) \Rightarrow (eT_i \in H))::(e'T_j \in H)$$

where eT_i is a transaction termination event for T_i (Section 3.3).

These could take the run-time form

On eT_j **do**

```
{x:= RootTi.TermStatus();
 if x = eTi then (eTj ∈ H) else (e'Tj ∈ H)
}
```

where $RootT_i$ is an interface with an operation TermStatus for discerning the termination status of transaction T_i.

This structure is an extension of the event-condition-action rule enunciated in [McCARTHY 89]. On the occurrence of the given event the if clause evaluates a condition and triggers the then clause (if the condition is true) or triggers the alternative else clause (if the condition is false). These On-if-then-else constructs are referred to as **event-based triggers**.

Event-based triggers (or, simply, triggers) do not fire unless they are active. Each trigger is activated by an *enabling mechanism* (Section 3.3.5) which specifies the name of the trigger and the transactions involved.

A *basic* trigger is automatically deactivated (disabled) the moment it fires. If such triggers are to be fired again they must be explicitly enabled again.

In contrast, a *perpetual* trigger, once enabled, remains active forever unless it is explicitly disabled (Section 3.3.5).

With this run-time structure in place, for example, the commit-on-abort (CAB) dependency defined in Section 3.3.2.3 would yield the following run-time form.

```
On commitTj do
        {x:= RootTi.TermStatus();
         if x = commitTi then (commitTj ∈ H) else (abortTj ∈ H)
        }
```

In this case, transaction T_j has requested to commit, resulting in the occurrence of the event $commitT_j$. Consequently, the termination status of the dependent transaction T_i is determined in order to commit T_j (if T_i aborted) or to abort T_j (if T_i committed).

Although the specific condition and action details will differ from one type of dependency to another, each type can be implemented as an event-based trigger that can be assembled during the process of constructing application-specific extended transactions (Section 3.7).

3.3.5 ETF operations for dependency management

The ETF infrastructure provides five operations for manipulating dependency controls.

- `DefineDependencyType (TypeSpecification)`: install a new dependency type
- `CreateDependency (`T_i`, `T_j`, Name:DependencyType, [perpetual])`: install a named dependency of the specified type between transactions T_i and T_j and make it a *perpetual* trigger (Section 3.3.4), if the `[perpetual]` option is specified.
- `DeleteDependency (`T_i`, `T_j`, Name:DependencyType)`: remove the named dependency between transactions T_i and T_j
- `EnableDependency (`T_i`, `T_j`, Name:DependencyType)`: enable the named dependency between T_i and T_j (i.e., enable the trigger for the dependency)
- `DisableDependency (`T_i`, `T_j`, Name:DependencyType)`: disable the named dependency between T_i and T_j (i.e., disable the trigger for the dependency)

The `Create`, `Enable`, `Disable` and `Delete (Dependency)` operations can each be used to specify a composite dependency, simply by presenting a list of dependencies, each of which relates transaction T_i to a set of other transactions. This list is treated as a conjunction of dependencies as described in Section 3.3.3.

3.4 Delegation specifications

An essential requirement of an extended transaction model is the ability of member transactions to delegate access[1] to some or all of their acquired object resources to other members.

In ACTA notation [CHRYSANTHIS 92], such delegation takes the form

$DelegateT_i$ (T_j, DelegateSet)

1. The term “access” is not to be confused with the use of this term in the context of privacy and security issues. The use herein refers to the controlled delegation of object resources among members of an extended transaction model.

where the transaction T_i delegates to another transaction T_j access to and responsibility for the object resources defined in DelegateSet.

Since rights of shared or exclusive access to object resources (object state) are acquired by invoking operations on the interfaces of objects, the Delegate function transfers to a specified transaction the responsibility for committing/aborting the object state that has been manipulated (accessed) by the delegator on the objects specified in the DelegateSet. This object state is committed/aborted by the delegatee, if and only if, it commits/aborts its execution without having further delegated responsibility for the same state to another transaction.

In the interest of flexibility, it is important to allow specific delegation to be triggered upon the occurrence of a particular transaction event. Formally, this is expressed as

$$(\text{Delegate}T_i\ (T_j, \text{Name:DelegateSet}) \in H) \Rightarrow (eT_i \in H)$$

with the meaning "transaction T_i delegates to transaction T_j the object resources specified in a named DelegateSet" if event eT_i is recorded in history H.

An explicit action on the part of T_j is required to complete the transfer of a named DelegateSet, specifically

$$\text{Acquire}T_j\ (T_i, \text{Name:DelegateSet})$$

The actual transaction events for which it is appropriate to pass shared object resources from one transaction to another depend on the underlying extended transaction model and its correctness criteria. A taxonomy of correctness criteria which focuses on relaxed serializability with respect to distributed system consistency requirements and transaction correctness properties is elaborated in [RAMAMRITHAM 92]. Specific principles and guidelines to assist the designer in formally specifying an extended transaction model and reasoning about its system and transaction correctness will be published in a later document, expanding the concepts and principles laid down in this technical overview of the proposed Extended Transaction Framework (ETF).

3.4.1 ETF operations for delegation management

The ETF infrastructure provides six operations for implementing object resource delegation controls.

- Create(Name:DelegateSet): create a named 'empty' DelegateSet (a trading context for transferring access to and responsibility for object resources from one transaction to another).
- Delete(Name:DelegateSet): delete the named DelegateSet.
- Insert(Name:ObjectResource, Name:DelegateSet): insert a named ObjectResource into the named DelegateSet.
- Remove(Name:ObjectResource, Name:DelegateSet): remove a named ObjectResource from the named DelegateSet.
- Delegate (T_i, T_j, Name:DelegateSet): transfer from transaction T_i to transaction T_j the object resources specified in the named DelegateSet.
- Acquire(T_j, T_i, Name:DelegateSet): transfer to transaction T_j from transaction T_i the object resources specified in the named DelegateSet.

These operations permit each DelegateSet to be created, manipulated and deleted dynamically during the execution of a transaction.

3.4.2 Example of delegation

The following example presents a scenario in which each figure is part of a continually developing story. In these figures, a horizontal solid line denotes a completed portion of a transaction's computation and its acquisition and manipulation of specific object resources. The arrowhead cursor labelled **t** indicates the present time, where time progresses from left to right on the horizontal axis.

Figure 3.3 represents two transactions, T_i and T_j, with a `begin-on-commit` dependency between them, such that T_j cannot begin until T_i has committed. At the current time, T_i has acquired and manipulated (read from and/or written to) three object resources O_1, O_2 and O_3 and has also inserted interfaces for these resources into the `DelegateSet` named `X`.

In Figure 3.4, T_i has issued a `Delegate` operation that transfers to T_j responsibility for accessing and committing/aborting the object resources, O_1, O_2 and O_3, given in the specified `DelegateSet` `X`. However, T_j cannot yet acquire these resources until it begins its execution, when and only when T_i commits.

Figure 3.3: Beginning of transaction T_i

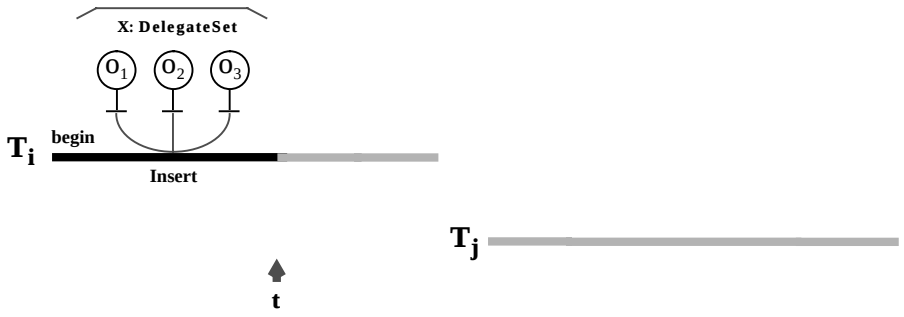


Figure 3.4: Delegation by transaction T_i to transaction T_j

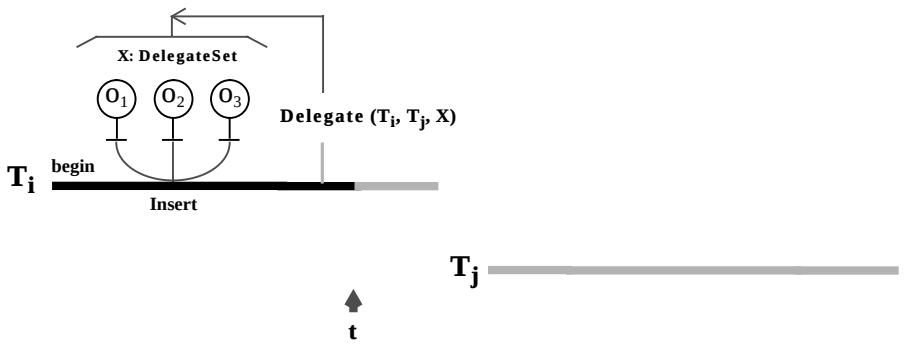


Figure 3.5 shows that T_i has committed, triggering T_j's execution, and thus enabling T_j to issue an `Acquire` operation to access `DelegateSet` `X`.

Finally, Figure 3.6 illustrates T_j manipulating objects O_1, O_2 and O_3 delegated to it by T_i.

Figure 3.5: Acquisition of delegateset by transaction T_j

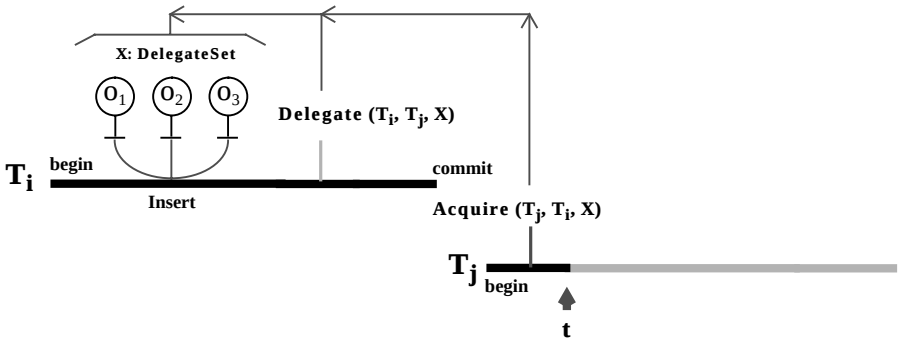
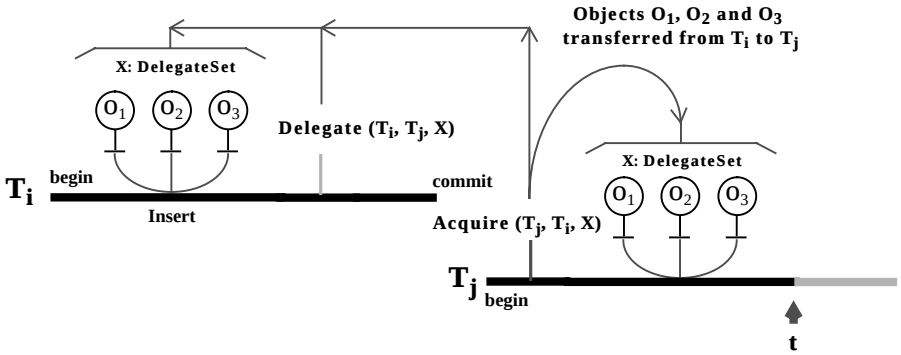


Figure 3.6: Transaction T_j in possession of a transferred delegateset



(Note that any effects on objects O_1, O_2 and O_3 will not be committed or aborted until transaction T_j commits or aborts, assuming, of course, that T_j does not subsequently delegate these resources to yet another transaction.)

3.4.3 Run-time aspects of delegation

In order to realize practical implementations of the delegation concept, it is necessary to provide a means by which delegation specifications can be mapped to corresponding engineering run-time components.

Accordingly, the specifications must be

1. mapped to specific application object resources
2. actioned at the appropriate execution points by transaction management operations.

A potentially promising technique for implementing the concept of delegation is the multi-coloured action model [SHRIVASTAVA 90]. The application of this model would allow each `DelegatSet` to be given a distinct colour. A transaction would then be allowed to acquire and access a chosen `DelegateSet` only if it possessed a token of the same colour.

3.5 Extended transactions in object-based interaction models

The foregoing has shown an extended transaction to define its component transactions and the relationships between them in terms of dependency and

delegation rules. These relationships thus specify the behaviour and structure of the extended transaction.

In object-based systems, the invocation of an object operation by one transaction member of an extended transaction results in the execution of another transaction member in the called object. In such cases, it is necessary to associate with the invocation those dependency and delegation rules that must be enforced between the invoking and invoked transactions.

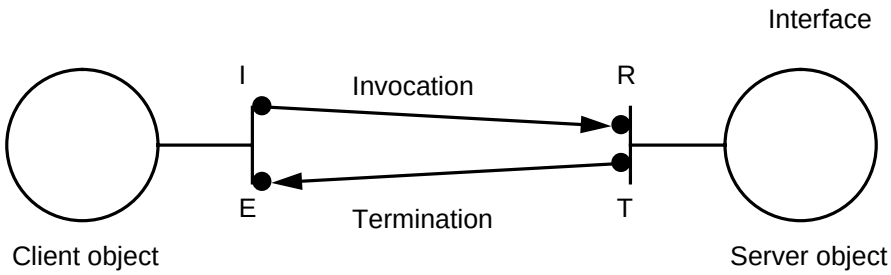
However, before considering how this can be specified, it is instructive to examine the nature of object-based interactions.

3.5.1 Object-based interaction model

The following short description of an object-based interaction model is taken from [EDWARDS 93].

The components of an ANSA and an ODP system are objects which interact through interfaces. The interaction allowed is described in [REES 93a] and [ODP 93] and is shown in figure 3.7.

Figure 3.7: The interaction model



A client object interacts with a server object by invoking operations on interfaces provided by the server.

Four events that occur in an interaction are illustrated in figure 3.7. Of these, two are events in which components engage, and the other two are observations of events made by components.

The event labelled I is the event in which the client engages to initiate the invocation. The value for this event consists of an operation name and zero or more parameters. The client's expectation will change when it engages in this event; it will be expecting a response that corresponds to the requested invocation.

The event labelled R is the server's observation of the invocation request. This is the "request event" described in [REES 93b] that informs the server that it has been invoked. The value of this event is expected to be the same as the value of the event I.

The event T is the "terminate event" described in [REES 93b] that occurs when the server has completed the evaluation of the operation. Its value consists of a termination name followed by zero or more parameters. The particular name and parameters are expected to be the ones defined by the behaviour of the

server given the value of the event R, the state of the server at the time of the event R, and any other events that may have been observed by the server.

The event E is the client's observation of the termination. It is expected to have the same value as the event T.

3.5.2 Interaction events, transaction dependencies and delegation rules

The ACTA notation permits the relationship between a client transaction, T_C, and its invoked server transaction, T_S, to be specified in terms of a postcondition of the invocation part (involving both events I and R) of the interaction shown in Figure 3.7:

$$post(invocationT_C[T_S]) \Rightarrow$$
$$((CreateDependency(T_C, T_S, D_X) \in H) \wedge (Delegate(T_C, T_S, DelegateSet_X) \in H)$$
$$\wedge (Acquire(T_S, T_C, DelegateSet_X) \in H))$$

This postcondition requires the underlying event ordering

$$CreateDependency(T_C, T_S) \rightarrow Delegate(T_C, T_S) \rightarrow I \rightarrow R \rightarrow Acquire(T_S, T_C)$$

Thus the client, before triggering the initiation event I of the invocation, first creates the required dependency (D_X) between it and the server and, secondly, delegates to the server those object resources it has in its possession which the server requires ($DelegateSet_X$).

When the server observes the invocation event R, it acquires the delegated object resources and then begins its expected service.

The postcondition for the termination part depicted in Figure 7.7, comprising events T and E, is

$$post(terminationT_S[T_C]) \Rightarrow$$
$$((Delegate(T_S, T_C, DelegateSet_Y) \in H)$$
$$\wedge (Acquire(T_C, T_S, DelegateSet_Y) \in H))$$

with the event ordering

$$Delegate(T_S, T_C) \rightarrow T \rightarrow E \rightarrow Acquire(T_S, T_C)$$

Before triggering the termination event T, the server delegates to the client those object resources in its possession which the client requires ($DelegateSet_Y$). The client, upon notification of event E, simply acquires these resources.

3.5.3 Dependencies and delegation with nested transactions

In the nested transaction model [MOSS 81], parent and child transactions interact in a hierarchy. When a parent invokes a child, it delegates to that child all the object resources that it and its ancestors have accessed[1]. Before a child aborts[2] or commits, those object resources delegated to it by its parent are returned to the parent. Moreover, if the child commits, any additional object resources it acquired during its execution are also delegated to the parent. However, the effects on all object resources accessed by all parent and child transactions are not made part of the permanent system state until the root transaction of the entire tree commits.

The characterization of an interaction between a parent (T_p) and a child (T_C) transaction is given below in terms of the postconditions of the invocation and the termination event sequences. In this example, the dependency (T_C CD T_p), or simply CD, refers to the commit dependency defined in Section 3.3.2.1, and

$AccessSet_{tree}$ refers to the object resources that are accessible to the parent and the child as these resources are passed up and down the tree.

The postcondition for an invocation of a child by the parent is

$$post(invocation T_p[T_c]) \Rightarrow$$
$$((CreateDependency(T_p, T_c, CD) \in H) \wedge (Delegate(T_p, T_c, AccessSet_{tree}) \in H)$$
$$\wedge (Acquire(T_c, T_p, AccessSet_{tree}) \in H))$$

with the underlying event ordering

$$CreateDependency(T_p, T_c) \rightarrow Delegate(T_p, T_c) \rightarrow I \rightarrow R \rightarrow Acquire(T_c, T_p)$$

The postcondition for a termination of the child to the parent is

$$post(termination T_s[T_c]) \Rightarrow$$
$$((Delegate(T_c, T_p, AccessSet_{tree}) \in H)$$
$$\wedge (Acquire(T_p, T_c, AccessSet_{tree}) \in H))$$

with the event ordering

$$Delegate(T_c, T_p) \rightarrow T \rightarrow E \rightarrow Acquire(T_p, T_c)$$

Satisfaction of these pre- and post- conditions throughout a nested transaction tree, together with the given commit dependency between each parent and its children, is sufficient to ensure a serializable execution of the tree as a whole.

3.6 Comment

The previous sections have focused on the modelling concepts of extended transactions. These concepts are the matters of process 1 of the complete ETF vision depicted in Figure 3.1. However, Figure 3.1 also depicts two further processes, namely, process 2 for the construction of application-specific extended transactions, and process 3 for defining the structure of business processes comprising several different application activities and their underlying extended transaction models. Although processes 1 and 2 are each to be the detailed subject of a future document, the present document would be incomplete without including a brief outline of the expected functionality of these processes and their relationship to each other.

---

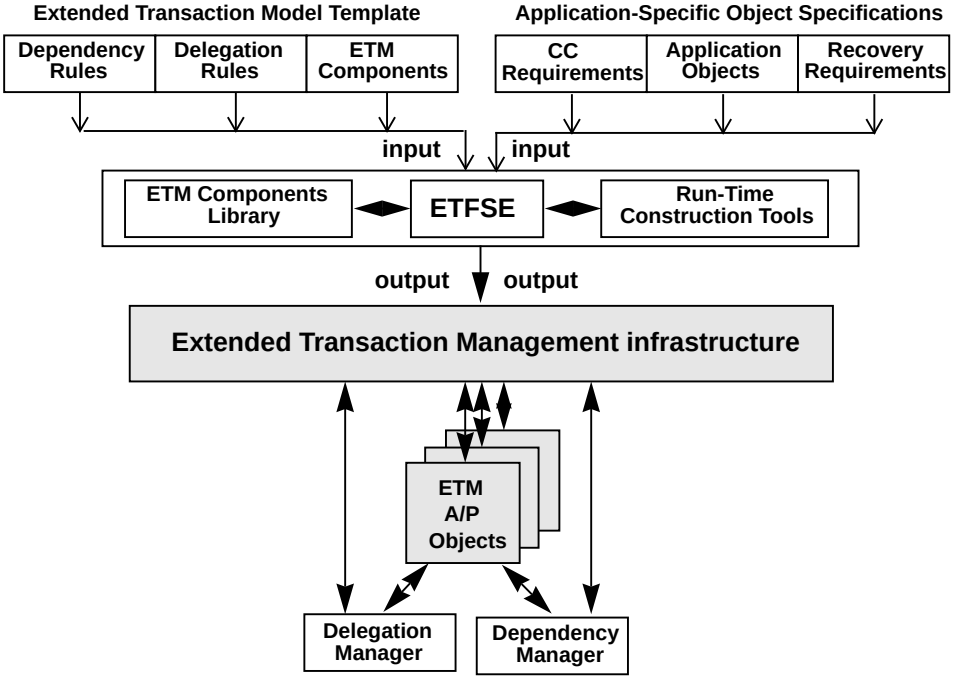
1. The nested transaction model considered here does not presume the possibility of the parent invoking concurrent children with overlapping object resource needs, which result in *read-write* or *write-write* dependencies [WARNE 93]. If such concurrency were permitted, the parent would not only need to delegate its acquired object resources to concurrent children, but also, in order to ensure a correct (execution consistent) outcome, require them to synchronize their actions such that they execute in some required planned application order whenever they attempt to access these resources. Note that [WARNE 93] fails to discuss the potential need for synchronizing parallel, sibling transactions in this way. However, this problem and its solution(s) will be addressed in a future ANSA document.

2. If a child transaction aborts involuntarily due to a crash failure, it will not be able to delegate any object resources to its parent. In such cases, the child requires a good samaritan (i.e., the ETF infrastructure) to perform this delegation on its behalf. Such failure semantics, including the orphan problem [PANZIERI 88, WARNE 93], is the subject of a future document.

3.7 Process 2: the construction of application-specific ETs

To support the construction of application-specific ETs, in accordance with the requirements of their models' formal specifications, the ETF provides an Extended Transaction Framework Support Environment (ETFSE). This environment comprises the components illustrated in Figure 3.8.

Figure 3.8: Process and components for constructing an application-specific extended transaction



For each extended transaction model, the ETFSE accepts a specification of the model in terms of its template. Each template defines the dependency and delegation rules for the model, together with a specification of the model's structure, expressed in terms of its transaction components and their run-time support transaction management operations (begin, abort, prepare, commit, etc.). Such transaction management operations have a corresponding run-time implementation in the specific ETMs components library contained in the ETFSE.

The application programmer interacts with the ETFSE to implement an application-specific variant of the model by supplying the specifics of the application, its object specifications, and its specific run-time concurrency control (CC) and recovery requirements. The ETFSE uses its run-time construction tools, the model's template specifications, and the application programmer's inputs to assemble and configure the run-time environment for the required extended transaction.

The resulting output of the ETFSE is an extended transaction comprising its application objects, its run-time extended transaction manager, and its dependency and delegation rule-based managers.

3.8 Process 3: the construction of transactional-based workflow schedulers

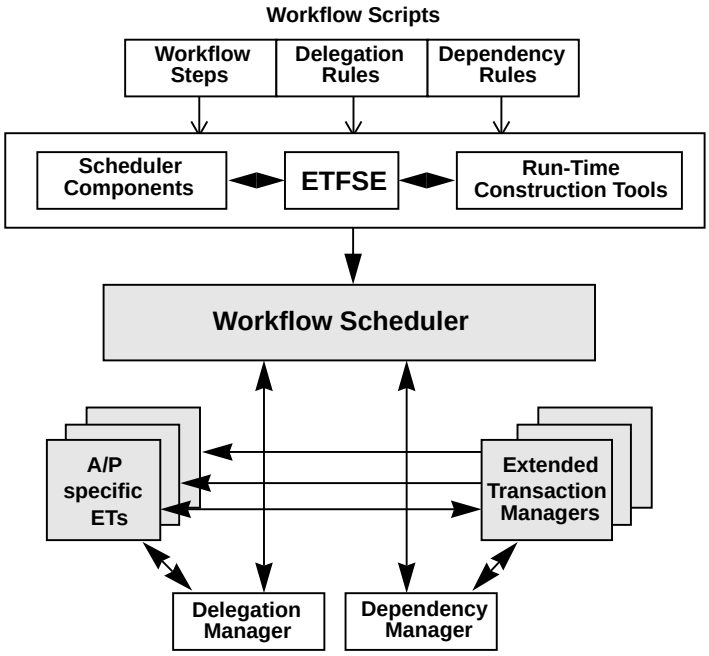
In the context of the ETF, a workflow is a collection of steps, each of which is the execution of an application-specific ET, organized to accomplish some business process. The scheduler for a workflow controls the order in which the steps are performed, the synchronization required among steps, and the flow of object resources between them. This scheduler is driven by a script which defines these controls.

As well as providing a support environment for construction of application-specific ETs, the ETFSE also supports the construction of workflow schedulers for controlling the execution of business workflows. The components of this construction process are depicted in Figure 3.9.

As shown, the ETFSE accepts scripts comprising three elements:

- workflow steps, each of which identifies the execution of a particular application-specific ET;
- dependency rules for controlling the order and synchronization of the workflow steps;
- the delegation rules which determine how overlapping object resources are to be shared among the workflow steps.

Figure 3.9: Process and components for constructing business workflows comprising several A/P specific extended transactions



The construction process transforms each workflow script into its run-time equivalent workflow scheduler. This transformation processes is effected by the ETFSE's run-time construction tools and its specialisation of the scheduler components with the script. The resulting workflow scheduler is itself executed as an extended transaction which interacts with its associated dependency and delegation managers to control the execution of the application (A/P) specific extended transaction (ET) steps, comprising the business process as a whole.

It is to be noted that each workflow scheduler can be arranged to serve as a global scheduler which acts as commit coordinator for its controlled application-specific ETs. In such cases, the coordinator would need to impose strong commit dependencies on these ETs and interact with them on their abort, prepare and commit intentions.

4 Review

This document has presented an overview of the modelling and construction processes and components of the proposed ANSA Extended Transaction Framework (ETF). The two principal modelling concepts of the architecture were presented: *dependency controls*, which specify the behavioural relationships between the member transactions comprising an extended (multi-transaction) model; and *delegation controls*, which specify how the member transactions can share access to object resources in a controlled manner. It was shown how the concepts could be used to describe the behaviour of the nested transaction model. The subsequent process for constructing an application-specific transaction based on a specific extended-transaction model and the process for then linking several extended transactions together to form a workflow were outlined.

4.1 Direction for future work

The following documents will be produced as a result of this document.

(1) Detailed specification of the ETF modelling concepts, including:

- complete and extended specification of the ACTA formal specification language and examples of its usage;
- basic set of widely applicable dependency rules and corresponding event-based trigger templates;
- detailed design of the delegation concepts in object-based environments with example implementations;
- ETF conformance to the ANSA naming, computational and engineering models.

(2) Methods and tools for constructing extended transactions, including:

- detailed design of ETF support environment;
- specification of transformer tools for assisting the application transaction construction process;
- specifications of run-time components libraries.

(3) Methods and tools for constructing workflow schedulers, including:

- specification of workflow script language;
- specification of transformer tools to assist run-time scheduler construction process;
- specification of scheduler run-time components library.

Following a complete specification of the ETF, it is projected that a suitable existing transaction environment will be selected to prototype the principles and mechanisms of constructing advanced extended transaction models and business workflows.

References

[BERNSTEIN 87]

Bernstein, P.A., Hadzilacos, V., Goodman, N., “Concurrency Control and Recovery in Database Systems”, Addison-Wesley Publishing Company Inc., 1989.

[CHRYSANTHIS 90]

Chrysanthis, P.K., Ramamritham, K., “ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior”, Proceeding of the ACM SIGMOD International Conference on Management of Data, 1990.

[CHRYSANTHIS 92]

Chrysanthis, P.K., Ramamritham, K., “ACTA: THe Saga Continues”, Database Transaction Models for Advanced Applications, Edited by Ahmed K. Elmagarmid, Morgan Kaufmann Publishers, 1992.

[DAYAL 90]

Dayal, U., Hsu, M., Ladin, R., “Organizing Long-Running Activities with Triggers and Transactions”, Proceeding of the ACM SIGMOD International Conference on Management of Data, 1990.

[EDWARDS 93]

Edwards, N.J., Rees, R.T.O., “A Model for Failures in Dependable Systems”, APM.1027, November, 1993, APM Ltd., Cambridge, U.K.

[ELMAGARMID 92]

Elmagarmid, A., K., (Editor), Database Transaction Models for Advanced Applications, Morgan Kaufmann Publishers, 1992.

[GARCIA-MOLINA 87]

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

[GEORGAKOPOULOS 92]

Georgakopoulos, D., Hornick, M., “An Environment for the Specification and Management of Extended Transactions and Workflows in DOMS”, TR-0218-09-92-165, October 1992, GTE Laboratories Incorporated.

[GRAY 93]

Gray, J., Reuter, A., “Transaction Processing: Concepts and techniques”, Morgan Kaufmann Publishers, 1993

[HEILER 92]

Heiler, S., Haradhvala, S., Zdonic, S., Blaustein, B., Rosenthal, A., “A Flexible Framework for Transaction Management in Engineering Environments”, Database Transaction Models for Advanced Applications, Edited by Ahmed K. Elmagarmid, Morgan Kaufmann Publishers, 1992.

[McCARTHY 89]

McCarthy, D.R., Dayal, U., “The Architecture of an Active Data Base Management System”, Proceeding of the ACM SIGMOD International Conference on Management of Data, 1989.

[MOSS 81]

Moss, J.E.B., “Nested Transactions: An Approach to Reliable Distributed Computing”, MIT/LCS/TR-260, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, U.S.A., 1981.

[NODINE 92]

Nodine, M.H., Ramaswamy, S., Zdonik., “A Cooperative Transaction Model for Design Databases”, Database Transaction Models for Advanced Applications, Edited by Ahmed K. Elmagamid, Morgan Kaufmann Publishers, 1992.

[ODP 93]

Basic Reference Model of Open Distributed Processing - Part 3: Prescriptive Model, Secretariat ISO/IEC JTC1/SC21, American National Standards Institute, June 1993.

[PANZIERI 88]

Panzieri, F., Shrivastava, S.K., “Rajdoot: A Remote Procedure Call Mechanism Supporting Orphan Detection and Killing”, IEEE Transactions on Software Engineering, Volume 14, Number 1, January 1988.

[PU 88]

Pu, C., Kaiser, G., Hutchinson, N., “Split Transactions for Open-Ended Activities”, IEEE Proceedings of the 14th Conference on VLDB, 1988.

[RAMAMRITHAM 92]

Ramamritham, K., Chrysanthis, P. K., “In Search of Acceptability Criteria: Database Consistency Requirements and Transaction Correctness Properties”, COINS Technical Report 92-54, Department of Computer Science, University of Massachusetts at Amherst, July 1992.

[REES 93a]

Rees, R.T.O.R., “ANSA Computational Model”, AR.001.01, APM Ltd., Cambridge U.K., April 1993.

[REES 93b]

Rees, R.T.O., “Using path expressions as concurrency guards”, TR.022.00, APM Ltd., Cambridge U.K., April 1993.

[RUSINKIEWICZ 93]

Rusinkiewicz, M., Sheth, A., “Specification and Execution of Transactional Workflows”, Bellcore Technical Memorandum, TM-STS-023284, August 1993.

[SHRIVASTAVA 90}

Shrivastava, S.K., Wheater, S.M., “Implementing Fault-Tolerant Distributed Applications Using Objects and Multi-Coloured Actions”, The 10th International Conference on Distributed Computing Systems, IEEE Computer Society, 1990.

[SHETH 93]

Sheth, A., Rusinkiewicz, M., “On Transactional Workflows”, Data Engineering Bulletin, 16 (2), June 1993.

[WARNE 93]

Warne, J.P., Rees, R.T.O.R., “ANSA Atomic Activity Model and Infrastructure”, APM Ltd., Cambridge U.K., January 1993.

[WHEATER 1990]

Wheater, S.M., “Constructing Reliable Distributed Applications using Actions and Objects”, Technical Report Series, No. 316, June 1990, Computing Laboratory, University of Newcastle upon Tyne.