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

Flexible Transaction Framework for Dependable Workflows

John Warne

Abstract

The success of an electronic market place for buying and selling information, services, and goods in an increasingly competitive world wide context will demand effective management tools to automate access to the many organisations, people, tasks and other resources that form the backbone of its routines, and to facilitate rapid change and/or reconfiguration of the services involved.

One such tool of emerging prominence is the workflow management tool!

Workflows are rule based management software that direct, coordinate and monitor multiple tasks arranged to form business processes for the rapid provisioning and acquisition of other services and related resources. Workflows put automatic rule enforcement into electronic business activities by controlling business processes using distributed system workflow supervisors.

If workflows are to be deployed effectively to control electronic businesses, they will need to operate with the ability to withstand system failures and continue to provide dependable service. Such dependability can be realised by adopting the operational characteristics of atomic transactions and consequently making workflows operate as dependable flexible transactions with built in mechanisms for failure detection and automatic backward/forward error recovery.

The flexible transaction framework presented in this document addresses these needs.

APM.1263.02 **Approved** 15 June 1994

Architecture Report

Distribution:
Supersedes:
Superseded by:

Flexible Transaction Framework for Dependable Workflows

Flexible Transaction Framework for Dependable Workflows

John Warne

APM.1263.02

14 June 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 specification of the proposed ANSA Phase III Flexible Transaction Framework (FTF): 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.

1.1 Business motivation

The necessity to buy and sell information, services, and goods in an increasingly competitive world wide context is inevitably paving the way for an electronic market place in which business processes can be rapidly deployed and enabled. Such business processes will be complex and widely distributed, and will typically involve several federated parties.

For example, envisage an electronic service which facilitates the purchase of a new car. This will involve several organisations, including the car dealer, the car manufacturer and those auxiliary organisations for putting in place the insurance and road tax necessary to allow the car to be put on the road. Moreover, not only will the car need to be paid for, but so will the electronic services which allow the purchase to occur. This payment process will also involve several parties. These parties will use different business procedures, forms, policies and technologies, all of which will need to be brought together to operate as an effective whole.

The success of such a market place and its diverse activities will demand effective management tools to automate access to the many organisations, people, tasks and other resources that form the backbone of its routines, and to facilitate rapid change and/or reconfiguration of the services involved.

One such tool of emerging prominence is the workflow management tool!

Workflows are rule-based management software services that direct, coordinate, and monitor multiple tasks arranged to form business processes for rapid provisioning and acquisition of other services and their related resources.

In short, workflows put automatic rule enforcement into electronic business activities by controlling business processes using distributed system workflow supervisors.

Workflows not only ensure orderly business practices, but also engender confidence in an organisation‘s management that the business is being properly controlled. This confidence induces management to allow delegation of responsibility and to do so with both additional confidence and control.

However, if workflows are to be deployed effectively to control electronic businesses, they will need to operate with the ability to withstand system failures and continue to provide dependable service.

Such dependability can be realised by adopting the operational characteristics (properties) of atomic transactions and consequently making workflows operate as flexible transactions with built in mechanisms for inter-service cooperation, failure detection and automatic forward and backward error recovery.

The vision projected is that of an electronic market place with flexible and dependable management controls for orchestrating transactional business application processes involving many diverse elements, all operating under the control of coordinating transactional workflows.

1.2 Challenges

The realisation of an effective transactional-based workflow management system naturally introduces several difficult challenges:

- Development of high level graphics tools for workflow construction, capable of use by management, technical and clerical staff.
- Development of flexible transaction facilities to permit workflows to be tailored precisely to the needs of an organisation and its business.
- Integration of workflow facilities in multi-media environments.
- Integration of workflows in real-time environments.
- Control of multiple workflows operating across federated system boundaries and varying QoS requirements.

Such challenges will need to be met if ever dependable federated systems of practical value to commerce and industry are to be put in place.

1.3 Risks

The main risk lies in not doing this work and thereby missing the opportunity to transform uncontrollable electronic business systems to organised rule-based havens. (“Go with the Flow”, Ruth Barrow, Infomatics, April 1994).

Moreover there is a risk of losing many lucrative consulting opportunities. Recent market research by OVUM estimates that workflow software revenues in Western Europe will rise from $144 million last year to $373 million by 1998. Early experience in transactional workflow systems promises substantial consulting opportunities. ((“Go with the Flow”, Ruth Barrow, Infomatics, April 1994). And this is only the tip of the iceberg, since this projection takes into account only a very small subset of the potential electronic market place, namely that of commercial office systems. It does not address the electronic service provisioning needs of insurance companies, travel agents, airline businesses, and the telecommunications industries, to name but a few!

1.4 Technical strategy

The strategy adopted herein is not simply to produce another transaction model, but rather to provide a unifying Flexible Transaction Framework (FTF) to facilitate the specification, construction, and inter-operation of different transaction models and workflow managers.

It is a primary goal that the FTF 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 FTF is to enable such application diversity to interoperate effectively.

1.5 Scope, audience and assumptions

The document as a whole scopes the applicability of dependable workflow management and the requirements for the supporting infrastructure. It provides a foundation for detailed design of specific workflow management systems. Accordingly, the document is targeted mostly at a technical designer audience. Moreover, it is assumed that this audience is 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 (e.g. [SHRIVASTAVA 90] and [WARNE 93]).

1.6 Relationship to other documents

This document supersedes document APM.1060.00.02: “Extended Transaction Framework: Technical Overview”. Much of the material of APM.1060 has been improved and incorporated into this document.

1.7 Document organisation

The remainder of the document is divided into eight chapters as follows.

Chapter 2 explains the need for flexible transactions. It reviews the properties of the traditional transaction model and highlights its benefits and its limitations. It then argues how these limitations can be overcome to enable the flexible transaction requirements of reactive, long-running, multi-party cooperative applications.

Chapter 3 provides a overview of the proposed Flexible Transaction Framework (FTF) in which flexible transactions are formed by collections of inter-dependent flat transactions with relaxed isolation properties. An outline of the architectural structure of the FTF is also given.

Chapter 4 defines the fundamental concepts and building blocks underpinning the architectural structure outlined in Chapter 3. It introduces the ACTA model as a framework for describing and reasoning about the structure and behaviour of flexible transactions. The principal theme of the chapter centres on two primary concepts; (1) *inter-transaction dependencies*, which constrain the execution patterns of flexible transactions, and (2) *delegation controls*,

which determine how the component transactions of flexible transactions can share resources in controlled ways.The chapter concludes with an illustration of how the concepts and their underlying mechanisms fit in the context of the ANSA object-based interaction model, with an example of how they can be applied to realise the behaviour of object-based nested transactions.

Chapter 5 summarises the general composition of a flexible transaction model and its structuring principles.

Chapter 6 outlines the processes and components needed to construct an application-specific flexible transaction and its run-time infrastructure.

Chapter 7 outlines how the FTF can be used to coordinate the execution of different flexible transactions as part of a dependable workflow, and how different workflows can be arranged to form a federated flexible transaction for controlling business processes.

Chapter 8 presents an example of dependable workflow management in a telecommunications services environment.

Chapter 9 concludes the document with a summary of the work and proposed future directions.

1.8 Acknowledgements

The author wishes to take this opportunity to express gratitude for the review comments on this document given by Jane Cameron (Bellcore/ANSA Team), Yves Caseau (Bellcore), Ian Domville (BNR Europe), Nigel Edwards (HP/ ANSA Team), Nancy Griffeth (Bellcore), Andrew Herbert (APM/ Technical Director and Chief Architect of the ANSA Project), Narayanan Krishnakumar (Bellcore), Rob van der Linden (APM/Research Manager and ANSA Team), Neil Mason (GPT), Tim Roberts (BNR Europe), Owen Rees (APM/ANSA Team), Santosh Shrivastava (University of Newcastle upon Tyne), Dave Stringer (BNR Europe) and Gomer Thomas (Bellcore/ANSA Team).

The author would also like to extend special thanks to Panos Chrysanthis (University of Pittsburg) and Krithi Ramamritham (University of Massachusetts, Amherst) and for their pioneering work on the ACTA meta-model for reasoning about transaction models in general; Marek Rusinkiewicz (University of Houston) and Amit Sheth (Bellcore) for their work on flexible transactional workflows; and Dimitrios Georgakopoulos (GTE Laboratories Incorporated) for his various work on transaction management platforms and related algorithms.

2 Need for flexible transactions

The benefits of using traditional ACID transactions to preserve 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 FTF 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) database applications.

The motivation for the FTF 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.

2.1 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.

2.2 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 FTF must provide architectural support for traditional transactions.

2.3 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 these resources without any one of them necessarily having to terminate (commit or abort) its execution. Moreover, the execution periods of such 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!

2.4 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 FTF addresses these concerns by providing a very flexible foundation for building application-specific models of any type.

2.5 Building on the work of others

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

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

2.5.2 Transaction-based workflow models

Transactional-based workflow models [DAYAL 90, SHETH 93, RUSINKIEWICZ 93, GEORGAKOPOULOS 92] are designed to enable collections of different flexible transactions to participate as sub-tasks of transactional-based, application-specific business processes.

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

This approach contrasts sharply with advanced workflow 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 a flexible 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 FTF is specifically intended to provide an environment in which application-independent, flexible transaction models can be combined with

application objects to generate application-specific, multi-transaction, business workflows.

3 Architectural summary

The main structuring principle of the FTF is that all flexible transactions and their underlying models are each formed by a collection of flat transactions with a set of execution dependencies between them and a set of rules which determine how they may share access to acquired resources.

Essentially, the set of transaction dependencies and rules for resource sharing for any specific flexible 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 flexible 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 a flexible 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.

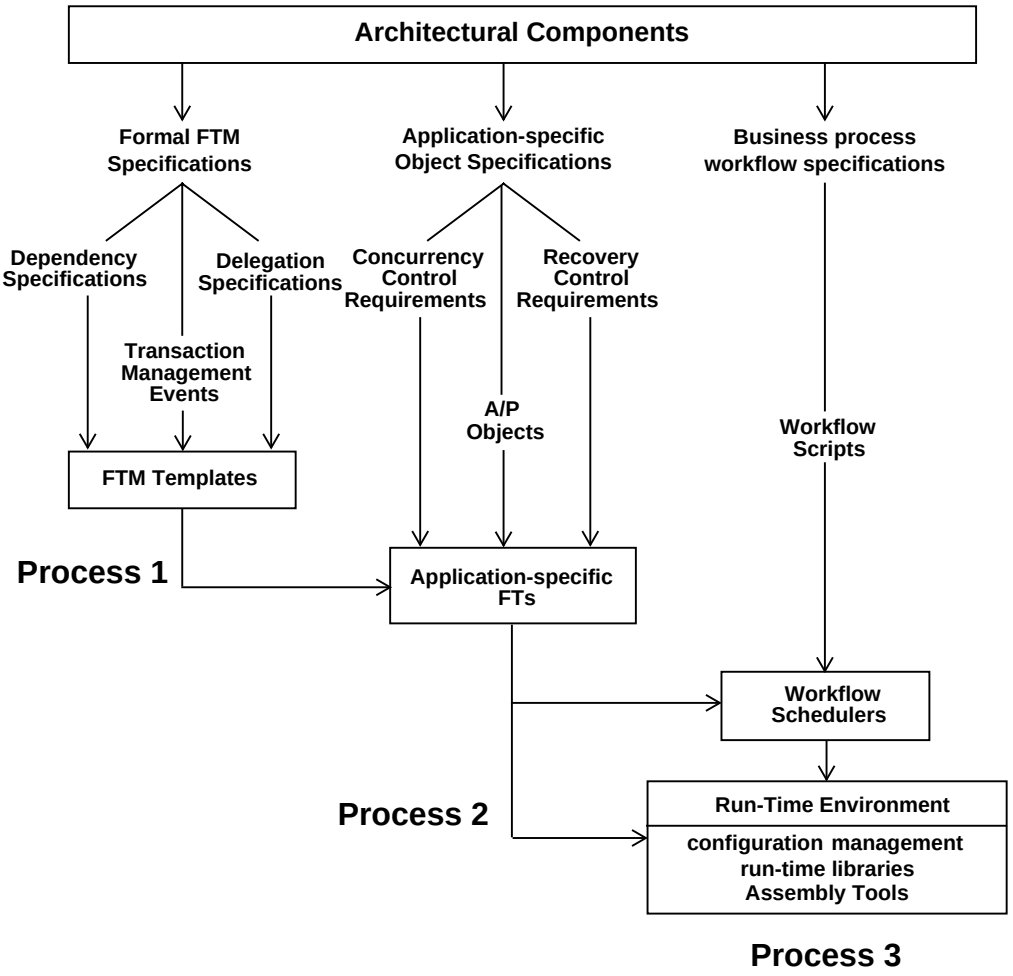
Different degrees of these attributes can be realized, firstly, by *relaxing the isolation property of member transactions* in order that they can observe each others computational effects; and, secondly, by *enforcing desired execution and system state access behaviours* through appropriate transaction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

3.1 Overall structure of the FTF

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

- **Process 1** constructs Flexible Transaction Model (FTM) templates using three elements derived from the formal specifications of the models:
  - — dependencies between member transactions of each model, which specify how the members interact and control each other' execution outcomes (e.g. transaction member T_j can only commit if transaction T_i commits)
  - — rules for specifying how member transactions may delegate access to shared resources among themselves
  - — specification of the transaction management events that identify the transaction management operations used to control the run-time behaviour of the member transactions.

Figure 3.1: Composition of the FTF



- **Process 2** constructs application-specific flexible transactions (FTs) by combining FTM templates (generated in process 1) with application (A/P) specific object specifications, a process which includes
  - — mapping dependencies and delegation specifications to application transactions and objects
  - — facilitating object-specific and transaction-specific concurrency and recovery controls
  - — assembling the supporting transaction management run-time infrastructure
- **Process 3** constructs workflow schedulers for controlling the execution of business processes defined by workflow scripts describing the run-time flow between and configuration of application-specific FTs generated during process 2, a process which includes
  - — mapping dependencies between different application-specific FTs
  - — generating a transactional workflow scheduler to execute the business process in accordance with the mapped dependencies
  - — creating the run-time multi-transaction infrastructure needed to support the scheduler and its controlled FTs

The three processes summarised above are elaborated in Chapters 5, 6 and 7 respectively. However, as a prerequisite, Chapter 4 introduces the concepts and basic building blocks for these processes.

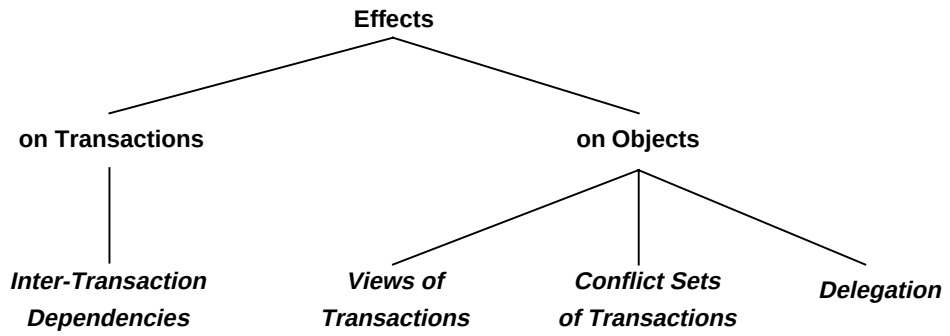
4 Fundamental concepts and building blocks

This chapter presents the concepts and mechanisms underpinning the three processes of the FTF summarised in Chapter 3. The concepts and the formal notation used to describe them are based on the ACTA meta-model [CHRYSANTHIS 92], a formal framework for describing and reasoning about flexible transaction models. The mechanisms illustrate how the concepts might be realised in a distributed object-based system.

4.1 Dimensions of the ACTA framework

ACTA characterises the effects of transactions on other transactions and the effects of transactions on the objects they access according to the taxonomy depicted in Figure 4.1.

Figure 4.1: Taxonomy of transaction effects



The notions expressed in this taxonomy are developed in the remainder of this chapter. However, some of the terminology and definitions given in the ACTA framework are modified to meet the specific requirements of the FTF.

4.2 Objects, interfaces, operation events, and object resources

ACTA (as does ANSA) assumes an object-based environment in which a system comprises distributed objects. A transaction accesses and manipulates the state of an object by invoking operations defined by one or more interfaces supported by that object. The state of an object is represented by its contents.

The invocation of an operation via an interface of an object is termed an *operation event*. Such events are further qualified as *application events* (when invoking operations on application objects) and *transaction management events* (when invoking transaction operations on the transaction management infrastructure).

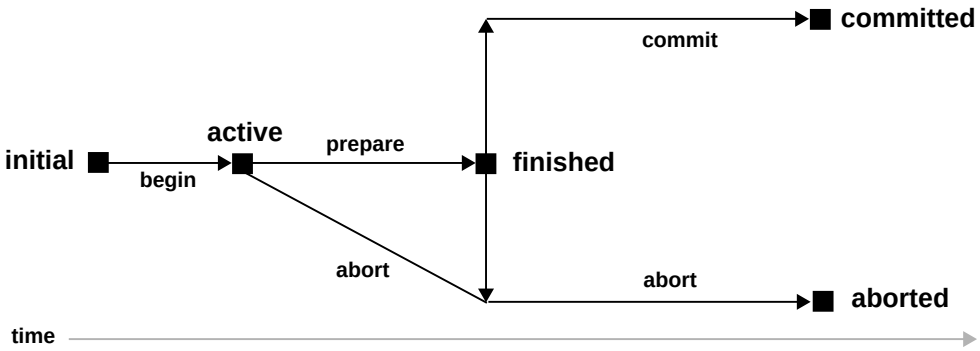
Throughout this document the term *object resource* refers to a piece of object state which is manipulated (read/written) by a transaction when it invokes an operation on an interface of an object.

4.3 Transaction state transitions

Just as each object has state which can be affected by transactions, so each transaction has state that can be affected by the occurrence of transaction management events. Such effects are called *transaction state transitions*.

The state transitions of a single transaction and the transaction management events that cause them are shown in Figure 4.2.

Figure 4.2: Transaction state transitions



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. From the `finished` state, a transaction can either `commit` (i.e., make the `committed` transition), or `abort` (i.e., make the `aborted` transition). The two transaction management 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 a flexible transaction model can regulate each others' state transitions and thus control each others' execution.

4.4 Histories and conditions on event occurrences

A *history* [BERNSTEIN 87] is a conceptual log for recording operational events. The concurrent execution of a set of transactions `T` can be represented by a history recording occurrences of both application events and transaction management events in the set `T`. The history presents a partial order in which these events occur.

The correctness properties of different transaction models can be expressed in terms of the histories produced by these models. The occurrence of an event in a history can be constrained in one of the following ways [CHRYSANTHIS 91]:

- an event `e'` can be constrained to occur only after another event `e`;
- an event `e` can only occur if a condition `c` is true;

- a condition c can require the occurrence of an event e.

These constraints lead to the following definitions:

- The predicate $(e < e')$ is true if event e precedes event e′ in some considered history H. It is false, otherwise.
- The expression $(e \in H) \Rightarrow Condition_H$ specifies that if event e exists in the history H then $Condition_H$ is true.
- $Condition_H \Rightarrow (e \in H)$ specifies that if $Condition_H$ is true, event e exists in the history H.

An additional definition is also useful

- $Condition_H \Leftrightarrow Condition_H'$ specifies that if $Condition_H$ is true then so is $Condition_H'$, and *vice versa*, in which case, the events satisfying these conditions will be recorded in the history H.

Expressions of the form $((e' \in H) \Rightarrow (e \in H))$ and $(e < e')$ are called *existence dependencies* and *ordering dependencies* respectively.

4.5 Inter-Transaction dependency specifications

A transaction dependency specification defines an inter-transaction relationship of the generic form

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

where T_j and T_i are transaction members of a given flexible transaction model which are related by a dependency of type x.

In its simplest form, each dependency type is expressed in terms of the occurrence of an event pair (eT_j, eT_i) and the relationship between these events pertaining to a transaction pair (T_j, T_i), where $T_i \neq T_j$.

In the following, an event is any transactional happening of interest (i.e., an invocation of a transaction management operation) that is permitted to belong to a conceptual execution history H, according to the constraints and definitions given in Section 4.4.

4.5.1 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 means “event eT_j can belong to history H only if event eT_i occurs and is recorded in H before event eT_j”.[1]

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

It is to be emphasised that such dependencies do not constrain T_j to generate event eT_j, even if event eT_i is generated and recorded in H.

1. Strictly, the meaning is “if eT_j exists in H then eT_i exists in H and eT_i precedes eT_j in the recording process. The less literal interpretation “can belong to” is used to denote that eT_i must exists in H before eT_j is allowed to exist in H. This less strict interpretation is used throughout the document to signify the need for determining the truth of a condition before allowing related events to be 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 "both events eT_i and eT_j can belong to history H provided both occur".

In this case, no specific ordering of these events is implied or required.

4.5.2 Alternative events

It is often necessary to express the requirement that a specific event be triggered (made to occur) should it be determined (or assumed) that some other event will never occur.

For *form 1* (Section 4.5.1), the required expression is

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

with the meaning "if it is determined that event eT_i will not occur, in which case event eT_j cannot belong to history H, then the alternative event $e'T_j$ is triggered (made to occur) and its occurrence is recorded in H.

Note that *form 1* expressions with alternative events are each derived from the logical conjunction of two basic *form 1* expressions:

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

If eT_j is a commit (or *resp*. an abort) event, the alternative $e'T_j$ is an abort (or *resp*. a commit event).

For *form 2* (Section 4.5.1), 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 $e'T_i$.

Form 2 expressions are each similarly derived from the logical conjunction of two basic *form 2* expressions.

(Note that alternative events are triggered by the FTF infrastructure on behalf of the transactions under consideration.)

4.6 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 FTF 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 future document.

4.6.1 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. Thus if T_j issues a prepare (to commit) operation it must await the termination outcome of T_i and then commit only if T_i commits. Of course, T_j can always abort no matter what T_i does.

For instance, the commit dependency is imposed between a parent T_i and its child T_j in the nested transaction model. The dependency is evaluated when the child enters its finished state by issuing a prepare (to commit) operation. If it is subsequently determined that the parent will not itself eventually commit, then the child must abort (or be aborted), thereby triggering the alternative event of the commit dependency.

The *commit dependency* is denoted by

$$(T_j \text{ CD } T_i)$$

4.6.2 Example 2: strong commit dependency

Transaction T_j has a strong commit dependency on transaction T_i:

$$((\text{commit}T_j \in H) \Leftrightarrow (\text{commit}T_i \in H))::(\text{abort}T_j, \text{abort}T_i \in H)$$

Both T_j and T_i commit, or both abort.

The strong commit dependency is used when it is necessary for both parties to have agreed execution outcomes as in the two-phase atomic commit protocol.

The *strong commit dependency* is denoted by

$$(T_j \text{ SCD } T_i)$$

4.6.3 Example 3: commit-on-abort dependency

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

$$((\text{commit}T_j \in H) \Rightarrow (\text{abort}T_i \in H))::(\text{abort}T_j \in H)$$

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.

For instance, envisage a customer requesting a travel agency to book a flight to a chosen destination preferably on British Airways (BA) or alternatively on Cathay Pacific (CP). The travel agent can trigger two transactions, T_i (for BA) and T_j (for CP), both in parallel and both executing under the constraint of the commit-on-abort dependency specified above. If T_i is successful and commits (in which case, T_j is aborted) the customer gets the preferred flight. If, however, T_i aborts, the customer either gets the alternative flight (if T_j is successful and commits) or no flight.

The *commit-on-abort dependency* is denoted by

$$(T_j \text{ COA } T_i)$$

4.6.4 Example 4: begin-on-commit-or-compensate dependency

Transaction T_j has a begin-on-commit dependency with transaction T_i which has a compensation dependency with T_c[1]:

$$(\text{begin}T_j \in H) \Rightarrow (\text{commit}T_i \in H)::(\text{commit}T_c \in H)$$

1. Transaction T_c is a *compensating transaction*. This means that its specific function is to semantically undo the commit action of the immediate predecessor of the transaction with which it shares a dependency, which in this case is T_i.

T_j can begin only if T_i commits, else a third transaction T_c is required to commit.

The begin-on-commit dependency (with an alternative action that requires a transaction T_c to commit if T_i does not commit) is required in the SAGA model [GARCIA-MOLINA 87] where the commit of one SAGA transaction member T_i triggers the next transaction member T_j in the SAGA sequence.

If, however, T_i aborts and was preceded by another transaction, say T_h, then a compensation transaction T_c must be triggered and committed to begin a backwards error recovery procedure on behalf of T_h.

The *begin-on-commit-or-compensate* dependency is denoted by

$$(T_j \text{ BOC } T_i \text{ CMP } T_c)$$

4.7 Composite dependencies

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

For *form 1* (Section 4.5.1) 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 4.5.1), 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 4.5.2). For example, the immediately preceding composite dependency would specify an alternative abort action for each of the transactions specified.

4.8 Effects of transactions on objects and induced dependencies

The notion of *conflict sets of transactions* in ACTA (Figure 4.1) is concerned with objects and the effects of transactions on objects resulting in conflicts between operations that induce dependencies between transactions.

Such induced inter-transaction dependencies are determined by the objects' synchronization properties. Broadly speaking, two operations are said to conflict if their effects on the state of an object or their return values are not independent of their execution order.

Serializability requirements induce the following dependencies between transactions invoking conflicting operations [CHRYSANTHIS 91]:

1. When the execution of an operation *q* follows an operation *p* on an object *ob* and both operations are *return-value-dependent*, the transaction T_j invoking operation *q* must abort *q* if for any reason the transaction T_i responsible for invoking operation *p* decides to abort *p*.

   Thus the two transactions share an abort dependency (AD), which in ACTA notation is expressed as

   $(return\text{-}value\text{-}dependent(p,q) \wedge pT_i[ob] < qT_j[ob]) \Rightarrow (T_j\ AD\ T_i)$

2. When the execution of an operation *q* follows an operation *p* on an object *ob* and both operations are *return-value-dependent*, the transaction T_j that invoked operation *q* cannot commit *q* unless the transaction T_i responsible for invoking operation *p* commits *p*.

   In this case, the two transactions share a commit dependency (CD), which in ACTA notation is expressed as

   $(return\text{-}value\text{-}dependent(p,q) \wedge pT_i[ob] < qT_j[ob]) \Rightarrow (T_j\ CD\ T_i)$

Awareness of these induced dependencies can be exploited to improve concurrent access to objects. To facilitate this, it is possible to associate with each object class a *compatibility table* that incorporates the semantics of the operations of the objects of that class. Specifically, (assuming object state is updated in place) the value of each (op_i, op_j) entry in an object's compatibility table will indicate one of four dependency values:

1. **OK**, meaning op_i and op_j are compatible and may execute concurrently;
2. **NOK**, meaning op_i and op_j are incompatible and may not execute concurrently and thus the commit or abort of one must precede the commit or abort of the other;
3. **AD**, meaning the execution of op_i and op_j is permitted with an induced *abort dependency* provided that op_j does not begin its execution until op_i has finished its execution and issued a prepare (to commit or abort) operation;
4. **CD**, meaning the execution of op_i and op_j is permitted with an induced *commit dependency* provided that op_j does not begin its execution until op_i has finished its execution and issued a prepare (to commit or abort) operation;

The compatibility table for an object can be derived by examining the semantics of the operations defined on that object [AGRAWAL 87].

4.9 Run-time structures for dependencies

Each inter-transaction dependency needs a run-time mechanism that enables it to be evaluated and actioned. The following two cases are instructive:

1. one transaction interrogates the status of another dependent transaction and thereafter triggers action appropriate to the response;
2. one transaction terminates its execution and triggers the execution of another dependent transaction.

Each case is considered below.

4.9.1 Case 1: transaction interrogates status of another dependent transaction

Consider dependency expressions of the form

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

where eT_i and eT_j are transaction termination events (Section 4.3).

These could take the imperative form (see Figure 4.2)

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

where $RootT_i$ is an interface with the 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 an event to request whether it is acceptable for a subsequent event eT_j to be recorded in the history H, 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 4.10) 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 for a given class of event occurrences unless it is explicitly disabled (Section 4.10).

With this run-time structure in place, for example, the commit-on-abort (COA) dependency defined in Section 4.6.3 would yield the following imperative form.

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

In this case, transaction T_j has requested to commit, via a prepare operation, naturally resulting in the occurrence of the event $prepareT_j$ in history H. 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).

4.9.2 Case 2: transaction triggers the execution of another dependent transaction

This case will arise, for example, when two transactions, T_i and T_j, are constrained by a "begin-on commit' (BOC) dependency, expressed as

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

Here, T_j can begin its execution only if T_i commits.

The following On-then imperative form would suffice.

```
On (commitTi ∈ H) then (beginTj ∈ H)
```

(Of course, if T_i were to abort then T_j would never spring to life.)

4.9.3 Note on dependency implementations

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 flexible transactions (Chapter 6).

4.10 FTF operations for dependency management

To support the specification and implementation of inter-transaction dependencies, The FTF infrastructure includes a component called the *dependency manager*. This manager provides an interface defining the following 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 4.8), 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 4.7.

4.11 Views of transactions

The *view* of a transaction specifies the objects visible to a transaction. As a transaction acquires object resource accessible through its *view*, it adds them to its *AccessSet*. Accordingly, a transaction's *AccessSet* specifies the distributed object resources for which it is responsible in the sense of having read from or written to these resources and therefore having responsibility to take commit or abort actions with respect to them.

4.12 Delegation specifications

An essential requirement of a flexible transaction model is the ability of its member transactions to delegate access[1] to some or all of their acquired object resources to other members. Such action permits a transaction to move responsibility for object resources from its *AccessSet* and allow another

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 a flexible transaction model.

designated transaction to assume responsibility for the same resources by moving them into its *AccessSet*.

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

$$\texttt{DelegateT}_i\,(\texttt{T}_j, \texttt{DelegateSet})$$

where the transaction $\texttt{T}_i$ delegates to another transaction $\texttt{T}_j$ access to and responsibility for the object resources defined in `DelegateSet`.

Since rights of shared or exclusive access to object resources 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

$$(\texttt{DelegateT}_i\,(\texttt{T}_j, \texttt{Name:DelegateSet}) \in \texttt{H}) \Rightarrow (\texttt{eT}_i \in \texttt{H})$$

with the meaning "transaction $\texttt{T}_i$ delegates to transaction $\texttt{T}_j$ the object resources specified in a named `DelegateSet`" if event $\texttt{eT}_i$ is recorded in history `H`.

An explicit action on the part of $\texttt{T}_j$ is required to complete the transfer of a named `DelegateSet`, specifically

$$\texttt{AcquireT}_j\,(\texttt{T}_i, \texttt{Name:DelegateSet})$$

The actual transaction events for which it is appropriate to pass shared object resources from one transaction to another depend on the underlying flexible 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 a flexible transaction model are summarised in Chapter 5.

4.13 FTF operations for delegation management

To support the specification and implementation of object resource delegation, The FTF infrastructure includes a component called the *delegation manager*. This manager provides an interface defining the following six operations for manipulating delegation controls.

- `Create(Name:DelegateSet)`: create a named 'empty' `DelegateSet` (a named 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 resource specified in the named `DelegateSet`.
- `Acquire`(T_j, T_i, `Name:DelegateSet`): transfer to transaction T_j from transaction T_i the object resource specified in the named `DelegateSet`.

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

4.14 Example of delegation

The following example presents a scenario in which each figure is part of a continually developing story. In the 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 4.3 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 object resources into the `DelegateSet` named `X`.

In Figure 4.4, 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 object resources until it begins its execution, when and only when T_i commits.

Figure 4.3: Beginning of transaction T_i

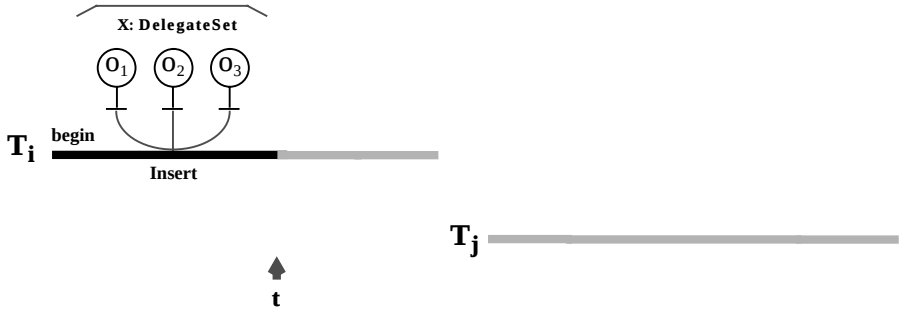


Figure 4.5 shows that T_i has committed, triggering T_j's execution, and thus enabling T_j to issue an `Acquire` operation to access `DelegateSet` `X`.

Finally, Figure 4.6 illustrates T_j manipulating objects O_1, O_2 and O_3 delegated to it by T_i.

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 object resources to yet another transaction.

This example represents a considerable oversimplification of a practical (real world) system, since it does not take account of failures that might occur to prevent transaction T_j from ever acquiring the delegated resources. If this were to happen, the delegated resources would become orphaned and would

Figure 4.4: Delegation by transaction T_i to transaction T_j

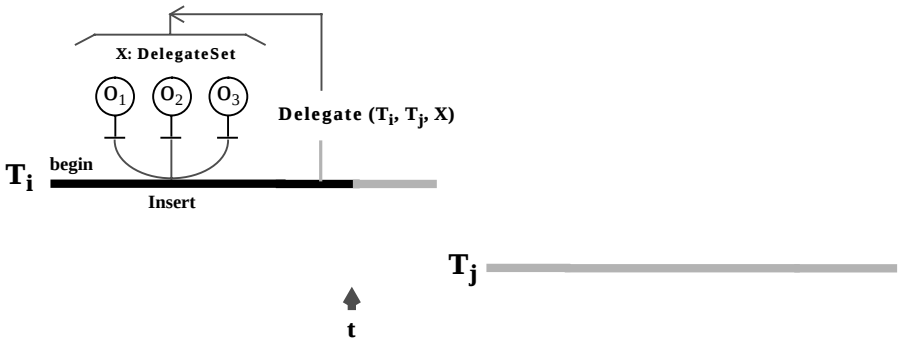


Figure 4.5: Acquisition of delegateset by transaction T_j

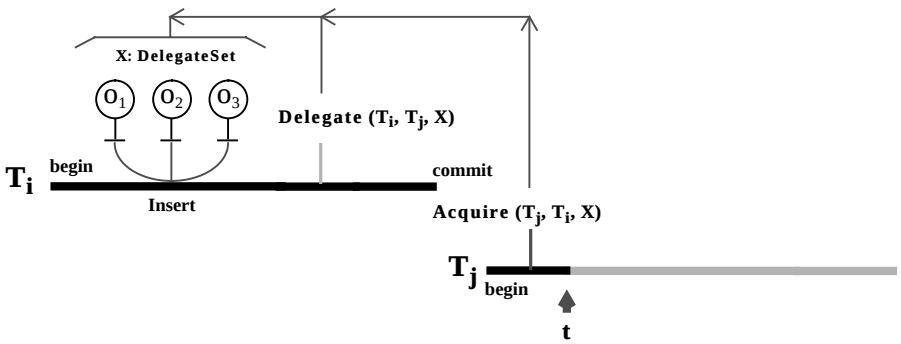
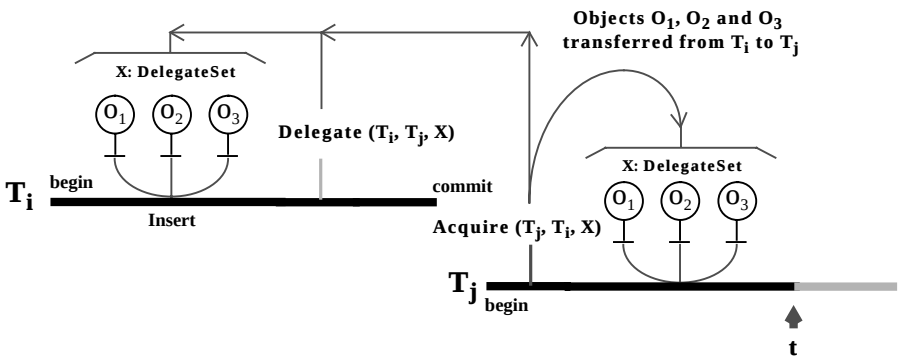


Figure 4.6: Transaction T_j in possession of a transferred delegateset



thus need to be recovered by the FTF infrastructure. This problem and its solution is similar to the problem of orphaned resources in nested transactions [WARNE 93].

4.15 Implementation notes on 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 `DelegateSet` 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. Perhaps each object resource should assume the colour of its `DelegateSet` as it is inserted. This approach suggests a method for passing coloured tokens between transactions via the `Delegate` and `Acquire` operations and suitably colouring the locks or timestamps applied to delegated object resources.

4.16 Flexible transactions in object-based interaction models

The foregoing has shown a flexible 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 flexible transaction.

In object-based systems, the invocation of an object operation by one transaction member of a flexible 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.

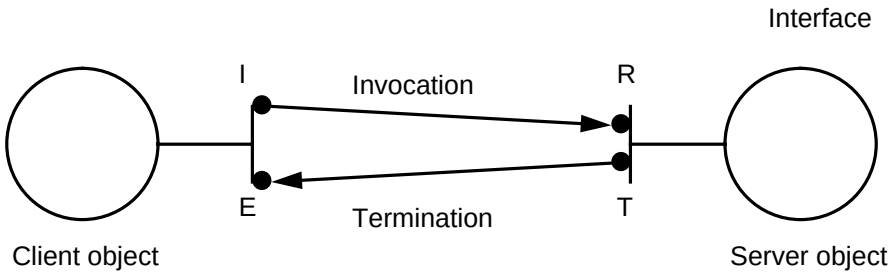
However, before considering how this can be specified, it is instructive to examine the nature of object-based interactions.

4.16.1 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 4.7.

Figure 4.7: 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 4.7. 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.

4.17 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 4.7:

$$post(invocationT_c[T_s]) \Rightarrow$$
$$((CreateDependency(T_c, T_s, D_x) \in H) \wedge (Delegate(T_c, T_s, DelegateSet_x) \in H)$$
$$\wedge (Acquire(T_s, T_c, DelegateSet_x) \in H))$$

This postcondition requires the underlying event ordering

$$CreateDependency(T_c, T_s) < Delegate(T_c, T_s) < I < R < Acquire(T_s, T_c)$$

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 of the termination part depicted in Figure 4.7, 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

$$Delegate(T_s, T_c) < T < E < Acquire(T_s, T_c)$$

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 object resources.

4.18 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 prepares to commit, those object resources delegated to it by its parent are returned to the parent. Moreover, if the child prepares to commit, any additional object resources acquired by it and its descendents (if any) are also returned 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 4.6.1, and $AccessSet_{tree}$ refers to the object resources that are accessible to the parent and the child as these resources are logically passed (inherited) up and down the tree.

The postcondition of an invocation of a child by its 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

$$CreateDependency(T_p, T_c) < Delegate(T_p, T_c) < I < R < Acquire(T_c, T_p)$$

The postcondition of a termination of the child to its 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

$$Delegate(T_c, T_p) < T < E < Acquire(T_p, T_c)$$

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 object 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 FTF 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.

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.

4.19 Comment

This chapter has focused on the modelling aspects of the Flexible Transaction Framework (FTF), based on the concepts of dependencies and delegation derived from the ACTA formal framework. Several practical mechanisms for realising these concepts were proposed.

The remaining chapters of the document review how these ideas can be used to develop flexible transaction models, application-specific flexible transactions, and transactional workflows corresponding to the three construction processes outlined earlier in Chapter 3.

5 Constructing flexible transaction models

This chapter summarises the general approach to be taken in the process of composing a flexible transaction model using the ideas presented in Chapter 4.

5.1 Starting with the flat atomic transaction model.

Since the basic building block for any flexible transaction model is the flat atomic transaction, it is instructive to review the properties of an atomic transaction and to do so in terms of its axiomatic definition.

The very much simplified definition given in Figure 5.1 is based on the more elaborate ACTA definition given in [CHRYSANTHIS 91].

Figure 5.1: Axiomatic definition of an atomic transaction (t)

1. $SE_t = \{Begin_t, Commit_t, Abort_t\}$
2. $(Begin_t \in H) \Rightarrow (\neg(Commit_t < Begin_t) \wedge \neg(Abort_t < Begin_t) \wedge \neg(Begin_t < Begin_t))$
3. $(Begin_t \in H) \Rightarrow (View_t = AccessSet_t \cup O_{system})$
4. t is failure atomic
5. $(Commit_t \in H) \Rightarrow ((Begin_t < Commit_t) \wedge \neg(Abort_t \in H))$
6. $(Abort_t \in H) \Rightarrow ((Begin_t < Abort_t) \wedge \neg(Commit_t \in H))$
7. t only invokes atomic operations
8. t is serializable

5.1.1 Axiom 1: significant events

This states that atomic transaction `t` is associated with three significant transaction management events: $\texttt{Begin}_t$, $\texttt{Commit}_t$ and $\texttt{Abort}_t$. The $\texttt{Begin}_t$ event signifies the instantiation of the transaction and the creation of the execution environment needed to ensure that its behaviour complies to the rest of its axiomatic definition.

The $\texttt{Commit}_t$ event denotes a successful outcome of the transaction, whereas the $\texttt{Abort}_t$ event denotes an unsuccessful outcome. The joint semantics of $\texttt{Commit}_t$ and $\texttt{Abort}_t$ complies to the ‘failure atomicity” semantics of Axiom 4.

5.1.2 Axiom 2: Instantiation property

This states that the $\texttt{Begin}_t$ event can be invoked at most once by `t` and must precede the invocation of either $\texttt{Commit}_t$ or $\texttt{Abort}_t$.

5.1.3 Axiom 3: view property

This restricts the visibility ($View_t$) of the atomic transaction to the object resources it possesses in its $AccessSet_t$ (initially empty when the transaction is instantiated) and any further object resources it can place in its $AccessSet_t$ by invoking operations on the interfaces of objects in the object system (O_{system}) in which it is operating.

5.1.4 Axiom 4: failure atomicity property

This informally stated axiom has a formal definition in ACTA which specifies axiomatically the required “all or nothing” semantics of an atomic transaction. One aspect of failure atomicity is captured by an “all” clause which states that a transaction commits if all the operations invoked by the transaction are committed. The other aspect of failure atomicity is captured by a “none” clause which states if an operation is aborted on an object, the invoking transaction must abort, and if the transaction aborts, all the operations it invoked are aborted.

5.1.5 Axiom 5: commit property

This states that only an instantiated atomic transaction can commit and that it cannot be committed after it has aborted.

5.1.6 Axiom 6: abort property

This states that only an instantiated atomic transaction can abort and that it cannot by aborted after it has committed.

5.1.7 Axiom 7: atomic object property

This specifies that all objects accessed by an atomic transaction comply to a serializable behaviour which ensures a serial execution of concurrent transactions which would otherwise incur operation execution conflicts.

5.1.8 Axiom 8: serializable property

This states that for an atomic transaction to behave correctly it must ensure that it operates in a serializable manner with respect to other concurrent atomic transactions.

The ACTA model has a complete and formal definition of serializable behaviour. Readers interested in the details of this definition and the other definitions cited above are referred to [CHRYSANTHIS 91, CHRYSANTHIS 92].

5.2 Comments on FTF development strategy

The following comments serve as a foreword to the structuring steps given in the next section.

The structuring steps for a developing flexible transaction model assume the existence of an atomic transaction capability (as defined above), together with the transaction management infrastructure needed to support its execution.

The proposed FTF is intended to be built by extending an existing transaction framework and not by developing an entirely new one from scratch. Moreover, the transaction platform chosen for extension would ideally support nested

transactions, with support for flat transactions as the degenerate case of the nested model.[1]

The transaction management extensions specifically required for the FTF are the inter-transaction dependency controls and object resource delegation operations defined in Chapter 4. An implementation of these extensions would require the basic transaction management infrastructure to interact with two additional management components: a dependency manager and a delegation manager.

5.3 General steps for developing a flexible transaction model

The steps given in this section assume that the axiomatic definition of the basic atomic transaction has been revised to include the significant events (SE_t), dependency events ($DepE_t$) and delegation events ($DelE_t$) shown in Figure 5.2.

Figure 5.2: Operation events for a flexible transaction model

SE_t = {Begin, Prepare, Commit, Abort}
$DepE_t$ = {DefineDependency, CreateDependency, EnableDependency,, DisableDependency, DeleteDependency}
$DelE_t$ = {Create, Insert, Remove, Delegate, Acquire, Delete}

These $DepE_t$ and $DelE_t$ operation events are defined in Chapter 4, Sections 4.10 and 4.13 respectively.

It is further assumed that this definition will be extended by the results of the steps given below and expressed using the ACTA formalisms. This will facilitate the designer in reasoning about the structure and semantics of the model.

The general steps taken in the process of developing a flexible transaction model will be as follows.

- Step 1: Define transaction membership

  This will define the number and names of the transactions comprising the model. In this sense, names are simply “local names” that are used to distinguish and refer to the transaction members in the context of the model.

  (Of course, the number and type of transaction members derived during this step will be wholly dependent on the nature of the flexible transaction model and its intended application domain. However, as a general statement, a number of very useful special transaction types have been defined (also specified in ACTA) which can serve as useful building blocks for the construction of a flexible transaction model. These types include

1. A suitable prototype of the FTF could be produced by extending the capabilities of the object-based ARJUNA transaction system developed at the University of Newcastle upon Tyne [SHRIVASTAVA 90]. Currently, ARJUNA supports nested transactions, class inheritance for refining transaction policies and mechanisms, and glued and coloured transactions to permit the flexible sharing of object resources [WHEATER 90]. Moreover, there are plans to develop ARJUNA on ORB-like platforms with CORBA compliance. All these features make ARJUNA a most attractive vehicle for implementing the flexible transaction extensions proposed in this document.

compensating transactions which logically undo the results of committed transactions whose effects are required to be removed, *contingency transactions* which can be executed in place of other transactions that fail, and *non-vital transactions* whose failures never result in aborting the flexible transactions of which they are members.

It is expected that these general transaction types will be usefully employed in the construction of many flexible transaction models.)

- Step 2: Define Inter-transaction dependencies

  This will define the execution flow of the flexible transaction by specifying the required inter-transaction dependencies between the named transaction members (as discussed in Chapter 4).

- Step 3: Define Inter-transaction delegations

  This will define all required delegation rules for those transaction members which participate in resource delegation (as discussed in Chapter 4).

The final step in the above process will require the model designer to implement the specific transaction management infrastructure mechanisms needed to support application-specific transactions based on the model.

Note that many flexible transaction models of different types will underpin common implementation mechanisms, including event based triggers (discussed in Chapter 4) for implementing inter-transaction dependencies. Moreover, the basic transaction management operations, as well as the dependency and delegation operations, will be common to all models. Thus once a library of common infrastructure mechanisms has been developed, the realisation of different application-specific flexible transactions should prove straightforward.

The process of constructing an application-specific transaction is outlined in Chapter 6.

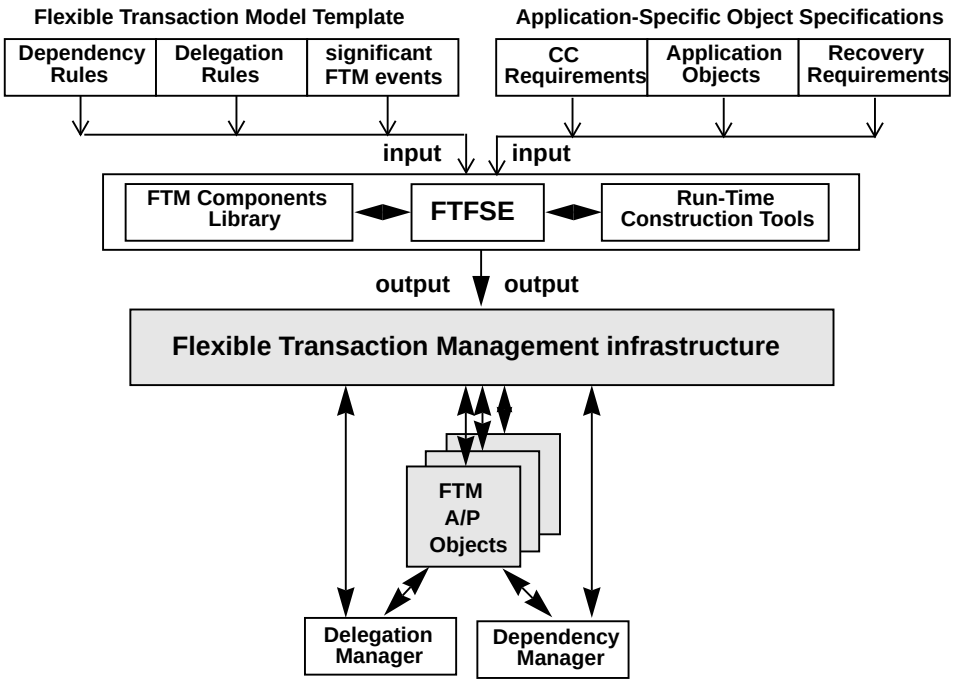
6 Constructing application-specific flexible transactions

This chapter builds on the ideas present in the previous chapters and outlines the process of constructing an application-specific flexible transaction and its resulting run-time infrastructure.

6.1 A flexible transaction framework support environment

To support the construction of application-specific FTs, in accordance with the requirements of their models' formal specifications, the FTF provides a Flexible Transaction Framework Support Environment (FTFSE). This environment comprises the components illustrated in Figure 6.1.

Figure 6.1: Process and components for constructing an application-specific flexible transaction



For each Flexible Transaction Model (FTM), the FTFSE 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, dependency and delegation controls have a corresponding run-time implementation in the specific FTM's components library contained in the FTFSE.

6.2 Implementation actions

The application programmer interacts with the FTFSE 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), as well as any specific recovery requirements (i.e., forward or backward recovery)

(Note that the concurrency control specifications are defined by the object-specific compatibility tables discussed in Chapter 4, which are supplied as part of the application-specific object specifications.)

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 flexible transaction in accordance with the dependency and delegation rules laid down by the corresponding FTM.

The resulting output of the FTFSE is a flexible transaction comprising its application objects, its run-time flexible transaction manager, and its dependency and delegation rule-based managers, tailored to the specific needs of the application.

7 Constructing transactional-based workflows

This chapter outlines the process envisaged for the construction of transaction workflows and their workflow schedulers.

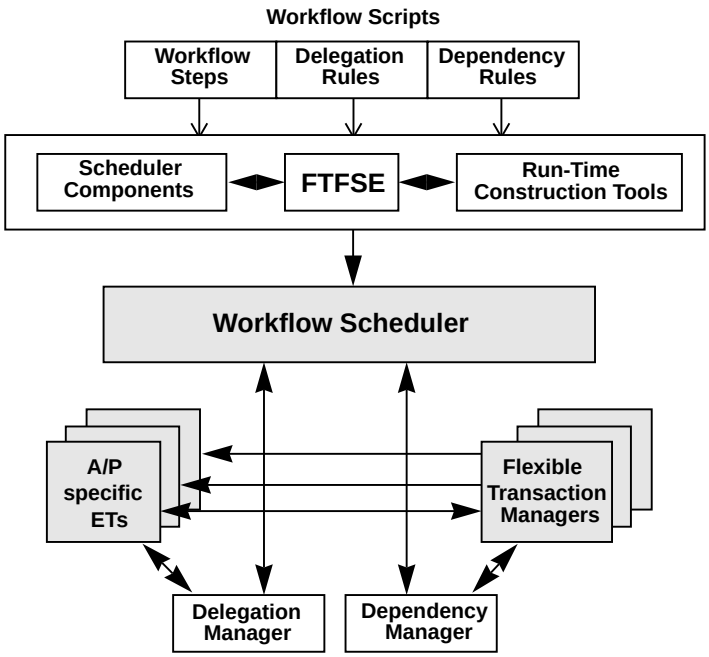
7.1 What is a workflow?

In the context of the FTF, a workflow is a collection of steps, each of which is the execution of an application-specific FT, 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.

7.2 Workflow construction process

As well as providing a support environment for construction of application-specific FTs, the FTFSE also supports the construction of workflow schedulers for controlling the execution of business workflows. The components of this construction process are depicted in Figure 7.1.

Figure 7.1: Process and components for constructing business workflows comprising several A/P specific flexible transactions



As shown, the workflow scripts describe three elements:

- workflow steps, each of which identifies the execution of a particular application-specific FT;
- 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.

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

7.3 Concluding comment and observation

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

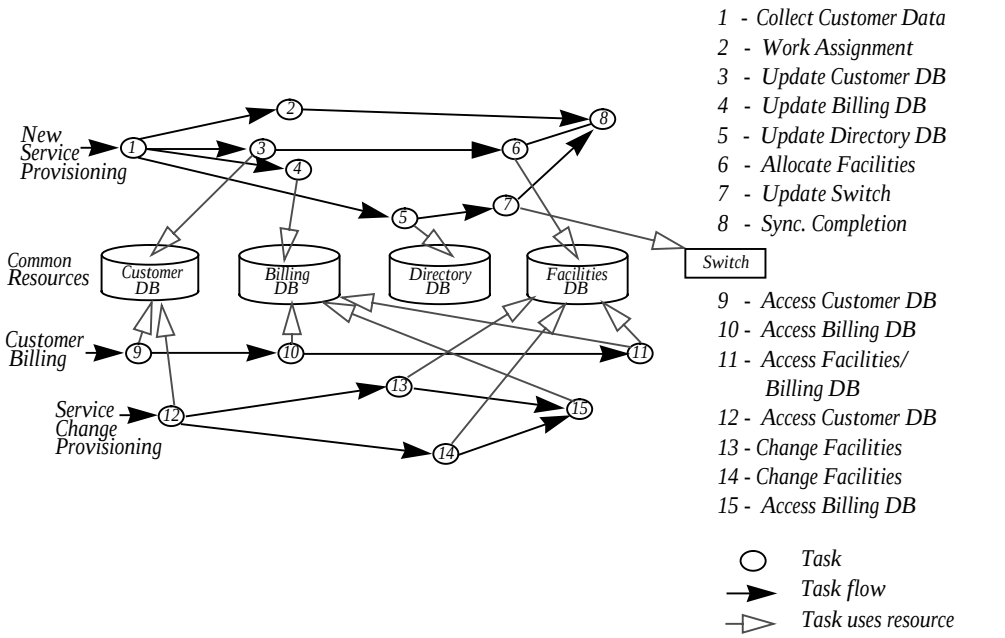
It is to be observed that the processes used for constructing a flexible transaction model and its application-specific transactions are similarly applied, albeit at a higher level of abstraction, for constructing dependable workflows and their application-specific schedulers.

8 Workflow example

This chapter presents an example of dependable workflow management in a telecommunications services environment. The example is borrowed from [GEORGAKOPOULOS 93] and discussed in the context of the proposed FTF.

Figure 8.1 depicts three concurrent telecommunications workflows, one for *new service provisioning*, another for *billing customers*, and a third for *service change provisioning*. These workflows access a combination of shared distributed resources, comprising databases and a switch.

Figure 8.1: Service provisioning with concurrent multiple workflows



It is supposed that all three workflows operate in an environment that supports the FTF.

To ensure dependable operation, each workflow functions as an application-specific flexible transaction. Thus each task within each workflow is itself a transaction and all such transactional tasks are related by inter-transaction dependencies and application-specific object resource delegation controls.

The following examines the organisation of one workflow in detail.

8.1 Service change provisioning

The purpose of the *service change provisioning* workflow is to modify the attributes of one or more available service facilities on a per customer basis. As illustrated in Figure 8.1, this function comprises four related transactional

tasks, namely, T12, T13, T14 and T15, the respective roles of which are outlined below.

- T12: Access Customer Database

  Since each instance of the workflow operates on a per customer basis, it needs to access the Customer Database to obtain each customer's record.

  (It is supposed that there is a directory of customer records that must be accessed first, followed by access to a desired record. If no changes are required with respect to customer details, these database elements can be accessed simply by reading them (i.e., accessed in read-mode only). If, however, changes are required, the directory is likewise read, but the customer record must be updated (i.e., accessed in read/write- mode.)

- T13 and T14: Change facilities

  For each customer, these tasks cooperate to change service facilities by updating the appropriate records in the Facilities Database. It is supposed here that T13 and T14 access different, but related, records in the Facilities Database for each change of customer service facility, thereby making it acceptable and efficient for these two tasks to execute in parallel.

- T15: Access Billing Database

  Finally, the customer is billed for the service change(s) by placing an invoice in the Billing Database.

The required inter-transaction dependencies for T12, T13, T14, and T15 are shown in Figure 8.2.

Figure 8.2: Inter-transaction dependencies for service change provisioning

Transactions	Dependency type	Meaning
T12, T13	T13 BOC T12	T13 can begin if T12 commits
T12, T14	T14 BOC T12	T14 can begin if T12 commits
T13, T14,T15	T15 BOC (T13 $\wedge$ T14)	T15 can begin if both T13 and T14 commit

Thus the transactional tasks of the workflow are related by begin-on commit dependencies, allowing T13 and T14 to begin in parallel if T12 commits, but T15 to begin only if T13 and T14 both commit.

It is essential that all tasks of the workflow commit their effects in order to complete the change of service process.

Since these tasks collectively update a number of databases, it is necessary to ensure that as each task commits, it delegates any updated resources to another uncommitted task in the flow.

For example, if T12 updates a customer record it must arrange for T15, say, to acquire that record and thus cause it to be committed/aborted when T15 commits/aborts. Similar provision must be made for the facilities records updated by T13 and T14.

In its role of acquiring uncommitted updates, T15 serves as a guardian to finally commit these effects for other terminated tasks, or to abort them otherwise.

It is to be noted that if T15 does not begin its execution (for any reason), then the effects of any updated resources delegated to it by an earlier task in the flow must eventually be aborted by the transaction management infrastructure. The infrastructure can easily deduce T15's destiny by observing the outcome of earlier dependent transactions in the workflow.

9 Summary and directions for future work

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

9.1 Direction for future work

It is proposed that the following work items be produced in response to the architectural framework presented in this document.

- Detailed design specification of the FTF, including
  - — complete specification of ACTA 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.
- Methods and tools for constructing flexible transactions, including:
  - — detailed design of FTF support environment;
  - — specification of transformer tools for assisting the application transaction construction process;
  - — specifications of run-time components libraries.
- 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.

To validate the FTF architecture and the above projected detailed design, it is proposed that the work is carried out with the support of an available transaction management platform which can be easily extended to prototype the ideas.

References

[AGRAWAL 87]

Agrawal, D., Carey, M.J., Livny, M., “Concurrency Control Performance Modeling: Alternatives and Implications”, ACM Transactions on Database Systems, 12(4), December 1987.

[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 91]

Chrysanthis, P.K., Ramamritham, K., “A Formalism for Extended Transaction Models”, Proceeding of the 17th International Conference on Very Large Data Bases, 1991.

[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.

[GEORGAKOPOULOS 93]

Georgakopoulos, D., Hornick, M., Manola, F., Brodie, M.L., Heiler, S., Nayeri, F., Hurwitz, B., “An Extended Transaction Environment for Workflows in Distributed Object Computing”, IEEE Bulletin of the Technical Committee on Data Engineering, Vol.16, No.2, June 1993.

[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., Dixon, G.N., Parrington, G.D., “An overview of ARJUNA: a programming system for reliable distributed computing”, Technical Report, 1990, Computing Laboratory, University of Newcastle upon Tyne.

[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.