Integrating Notifications and Transactions: Concepts and X^2TS Prototype

C. Liebig, M. Malva, A. Buchmann

Database and Distributed Systems Research Group
Darmstadt University of Technology

Abstract. Event-based architectural style promises to support building flexible and extensible component-oriented systems and is particularly well suited to support applications that must monitor information of interest or react to changes in the environment, or process status. Middleware support for event-based systems ranges from peer-to-peer messaging to message queues and publish/subscribe event-services. Common distributed object platforms restrict publishing events on behalf of transactions to *message integrating transactions*. We suggest that concepts from active object systems can support the construction of reliable event-driven applications. In particular, we are concerned with unbundling transactional reactive behavior in a CORBA environment and introduce X^2TS as integration of transaction and notification services. X^2TS features rich coupling modes that are configured on a per subscriber basis and supports the application programmer with coordinating asynchronous executions on behalf of transactions.

1 Introduction

Messaging and event-services are attractive for building complex component-oriented distributed systems [16, 22, 17]. Moreover, systems designed in an event-based architectural style [9] are particularly well suited for distributed environments without central control, to support applications that must monitor information of interest or react to changes in the environment, or process status. The familiar one-to-one request/reply interaction pattern that is commonly used in client/server systems is inadequate for systems that must react to critical situations and exhibit many-to-many interactions. Examples for such applications are process support systems and workflow management systems. They are best constructed using event-based middleware, realizing the control flow and inter-task dependencies by event-driven task managers [28]. In systems like weather alerts, stock tracking, logistics and inventory management, information must be disseminated from many publishers to an even larger number of subscribers, while subscribers must be given the ability to select and extract the data of interest out of a dynamically changing information offer [32, 4]. In related research as well as in de-facto standard message oriented middleware, reliability concerns are restricted to event-delivery using transactional enqueue/dequeue or transactional publish/subscribe [49]. Only minimal support is given to structure the dependencies between publisher and subscriber with respect to their transaction context. In distributed object systems, transactions are a commonly used concept to provide the simple all-or-nothing execution model. Transactions bracket the computations into spheres of atomicity. When using an event-based style this implies asynchronous transaction branches and dynamically evolving structure of atomicity spheres. Dependencies between publisher and subscriber can be expressed in terms of visibility - i.e. when should events be notified to the subscribers - in terms of the transaction context of the reaction and in terms of the commit/abort dependencies between the atomicity spheres of action and reaction.

The overall goal of the *Distributed Active Object Systems* (DAOS) project is to unbundle the concepts of *active object systems* [7], which are basic to active database management systems. An active object autonomously reacts to critical situations of interest in a timely and reliable manner. It does so by subscribing to possibly complex events that are detected and signalled by the active object system. To provide reliable and correct executions the action and reaction may be transactionally coupled. As active functionality is useful beyond active databases, we aim to provide configurable services to construct active object systems on behalf of a distributed object computing platform, CORBA in particular. While DAOS in general addresses many of the dimensions of active systems as classified in [50, 44, 14], in this paper we focus on X^{2}TS, an integration of notification (NOS) [37] and transaction service (OTS) [40] for CORBA. X^{2}TS realizes an event-action model which includes transactional behavior, flexible visibilities of events and rich coupling modes. The prototype leverages COTS publish/subscribe MOM to reliably distribute events. The prototype deals with the implications of asynchronous reactions to events. We explicitly consider that multiple subscribers with diverse requirements in terms of couplings need to be supported. The architecture also anticipates to plug in composite event detectors.

The rest of this paper is organized as follows. In the next section we will briefly discuss related work. In Section 3 we will introduce the concept of coupling modes and the dimensions for a configurable service. We discuss the application of active object concepts to event-based process enactment. In Section 4, we will present the architecture and implementation of the X^{2}TS prototype. Section 5 concludes the paper.

2 Related work

Work has been done on integrating databases as active sources of events and unbundling ECA-like rule services [18, 27] for distributed and heterogeneous environments. Various publish/subscribe style notification services have been presented in the literature [35, 33, 10, 20, 47]. They focus on filtering and composing events somewhat similar to event algebras in active DBMSs [15, 41]. In [49], the authors identify the shortcomings of transactional enqueue/dequeue in common message oriented middleware and suggest message delivery and message processing transactions. The arguments can be applied to transactional publish/subscribe [12], as well. Besides the work in [49], to our knowledge the provision of publish/subscribe event services that are transaction aware has not been considered so far. A variety of process support and workflow management systems are event-driven - or based on a centralized active DBMS - and encompass execution models for reliable and recoverable enactment of tasks or specifying inter-task dependencies [53, 21, 25, 28, 13, 11]. However those systems are merely closed and do not provide transactional event-services per se. The relationship of a generic workflow management facility to existing CORBA services and the reuse and the integration thereof needs further investigation [46,8].

3 Notifications & transactions: couplings

In a distributed object system, atomicity of computations can be achieved using a two-phase commit protocol as realized by the CORBA OTS for example (see section 4.1 for a brief introduction to OTS). All computations carried out on behalf of an OTS transaction are said to be in an atomicity sphere. Atomicity spheres may be nested, but still are restricted to OTS transaction boundaries. OTS transactions relief the software engineer to deal with all possible error situations that could arise to the various failure modes of participating entities: either all computations will have effect, or none of them will (all-or-nothing printciple). If isolation of access to shared data and resources is an issue, a concurrency control mechanism must be applied, typically two-phase locking as in the CORBA Concurrency Service. As a consequence, the isolation sphere corresponds to the transaction boundaries. Objects may be stateful and require durability. In that case full fledged ACID transactions are appropriate.

The design issue arises, where to draw the transaction boundaries. There are trade-offs between transaction granularity and failure tolerance: always rolling back to the beginning is not an option for long running computations. On the other hand, simply splitting the work into smaller transactional units re-introduces a multitude of failure modes that must be dealt with explicitly. In addition, there are trade-offs between transaction granularity and concurrent/cooperative access to shared resources and consistency. Long transactions reflect the application requirements as they isolate ongoing computations, hide intermediate results and preserve overall atomicity. However, common concurrency control techniques unnecessarily restrict cooperation and parallel executions. Short transactions allow higher throughput but might also commit tentative results.

3.1 Coupling modes for event-based interaction

When looking at event-based interactions, a variety of execution models are possible. If we take reliability concerns into account, the consumer's reaction to an event notification can not considered to be independent of the producer in all cases. For example, dependent on the need of the event consuming component, one could deliver events only if the triggering transaction has commited. In other cases, such a delay is inacceptable. The reaction could even influence the outcome of the triggering transaction or vice versa. Therefore the notion of a coupling mode was introduced in the HiPAC [15] project on active database management systems. Coupling modes determine the execution of triggered actions relative to the transaction in which the triggering event was published [7,5]. X^2TS provides a mechanism to reliably enforce flexible structures of atomicitiy spheres between producers and consumers through the provision of coupling modes.

We consider the following dimensions that define a coupling mode in X^{2}TS:

- *visibility*: the moment when the reacting object should be notified
- *context*: the transaction context in which the object should execute the triggered actions
- *dependency*: the commit(abort)-dependency between triggering transaction and triggered action
- *consumption*: when the event should be considered as delivered and consumed

Table 1 shows the configurable properties assuming a flat transaction model. We have also considered the possible coupling modes in case of nested transactions. Because of the arbitrary deep nesting of atomicity spheres, many more coupling modes are possible. Yet, not all of them are useful, as transitive and implicit dependencies must be considered. For sake of simplicity, they are not discussed here and will be presented elsewhere.

visibility	immediate, on commit, on abort, deferred
context	shared, separate top
forward dependency	commit, abort
backward dependency	vital, mark-rollback
consumption	on delivery, on return, explicit, atomic

Table 1: Coupling modes

- With *immediate* visibility, events are visible to the consumers as soon as they arrive at the consumer site and independent of the triggering transaction's outcome. In case of *on commit* (*on abort*) visibility, a consumer may only be notified of the event if the triggering transaction has committed (aborted). A *deferred* visibility notifies the consumer as soon as the event producer starts commit processing.
- A *commit* (*abort*) forward dependency specifies, that the triggered reaction commits only if the triggering transaction commits (aborts).
- A backward dependency constrains the commit of the triggering transaction. If the reaction is *vital* coupled, the triggering transaction may only commit if the triggered transaction has been executed and completed successfully.
  If the consumer is coupled in *mark-rollback* mode, the triggering transaction is independent of the triggered transaction commit/abort but the consumer may explicitly mark the producer's transaction as *rollback-only*.
  Both backward dependencies imply, that a failure of event delivery will cause the triggering transaction to abort.
- Once an event has been consumed, the notification message is considered as delivered and will not be replayed in case the consumer crashes and subsequently restarts. The event may be consumed simply by accepting the notification (*on delivery*) or when returning from the reaction (*on return*). Alternatively, consump-

tion may be bound to the commit of the consumer's atomicity sphere (*atomic*) or be *explicit*ly indicated at some point during reaction processing.

If the reaction is coupled in a *shared* mode, it will execute on behalf of the triggering transaction. Of course this implies a forward and backward dependency, which is just the semantic of a sphere of atomicity. Otherwise, the reaction is executed in its own atomicity sphere, i.e. *separate top* and the commit/abort dependencies to the triggering transaction can be established as described above.

3.2 Discussion

The most notably distinctions between coupling modes in X^2TS and in active databases as proposed in [7,5] are that

i) coupling modes may be specified per event type even across different transactions

ii) X^2TS supports parallel execution in the shared (same) transaction context

iii) publishing of events is non-blocking, i.e. blocking execution of triggered actions, is not supported

Backward dependencies even if implicit, as in ii) - raise the difficulty of how to synchronize asynchronous branches of the same atomicity sphere. This is why we introduce checked transaction behavior. A triggering transaction may not commit before the reactions that have backward dependencies are ready to commit (and vice versa). Publish/subscribe in principle encompasses an unknown number of different consumers. Therefore, we require that the consumers, that may have backward dependencies and which need to synchronize with the triggering action, be specified in a predefined group.

With the same arguments as presented in [43], event composition may span several triggering transactions and we support couplings with respect to multiple triggering transactions.

The visibility modes suggested, provide more flexibility than transactional messaging (and transactional publish/subscribe). Transactional messaging, as provided by popular COTS MOM, only focuses on message integrating transactions [49], that is messages (events) will be visible only after commit of the triggering transaction. This restricts the system to short transactions and therefore provides no flexibility to enforce adequate spheres of atomicity (and isolation). Also, flexible and modular exception handling - as supported by abort dependencies in our system - is not possible.

On the other hand, principally ignoring spheres of atomicity - using transaction unaware notifications - is not tolerable by all applications: with *immediate* visibility, computations are externalized (by publishing events) that might be revoked, later. With respect to serialization theory [2] publishing an event can be compared to a write operation, while reacting to it introduces a read-from relation. Therefore dirty reads - reactions to immediate visible events of not-yet-committed transactions - may lead to non recoverable reactions or cascading aborts. While in some cases it may be possible to revoke and compensate an already committed reaction, in other cases it might not be. The design of X^2TS considers the fact, that a single execution model will not be flexible enough to fulfil the diverse needs of different consumers. Therefore, coupling modes, most important visibility, are configured on a per consumer basis. While one consumer may be restricted to react to events *on commit* of the triggering transaction, others may need to react as soon as possible and even take part in the triggering transaction e.g. to check and enforce integrity constraints. More loosely couplings may be useful, too. For example an active object that reacts immediately without a forward commit dependency but additionally registers an abort dependent compensating action, in case the triggering transaction eventually aborts.

An event-based architectural style is characterized to exhibit a *loosely coupled* style of interaction. We suggest that for the sake of reliable and recoverable executions in event-based systems, atomicity spheres and configurable coupling modes should be supported. Nevertheless, X^2TS provides the advantages of publish/subscribe many-to-many interactions with flexible subject based addressing in contrast to client-server request reply and location based addressing. The asynchronous nature of event-based interactions is sustained as far as possible and the differing demands of heterogeneous subscribers are taken care of.

3.3 Example: event-based process enactment

Separation of concerns is the basis for a flexible and extensible software architecture. Consequently, one of the principles of workflow and process management - separating control flow specification and enforcement from functionality specification and implementation - is a general design principle for component-oriented systems [1,3,29,45]. Flow independent objects, that implement business logic, are invoked by objects that reify the process abstraction, i.e. process flow knowledge. In the following, we will briefly characterize the principles of event-based process enactment and the application of transaction aware notifications.

A process model (workflow schema) is defined using some process description language to specify activities that represent steps in the business process. Activities may be simple or complex. A complex activity contains several subactivities. Activities must be linked to an implementation, i.e. the appropriate business object. Activities are connected with each other using flow control connectors like split and join. In addition event connectors should be provided to support inter-process dependencies and reactive behavior at the process model level [21]. An organizational model and agent selection criteria must be specified, as well.

The process enactment system at runtime instantiates for each activity a task and a task manager. The task manager initiates the agent assignments and business object invocations, keeps track of the task state, handles system and application level errors, verifies the pre- and postconditions and enforces control flow dependencies. In any case, a protocol must be established for the interaction between task manager and business objects as well as task manager and other process enactment components (i.e. other task managers, resource managers, worklist handlers).

In the case of an event-based architecture, a task manager subscribes and reacts to events originating from tasks (i.e. activity instantiations), other task managers, resource and worklist managers as well as from external systems. This requires the participating components to externalize behavior and publish events of interest or allow instrumentation for event detection. Task state changes are published as events, for example "task creation", "task activation"; "task completion". The task manager reacts to error situations, task completions, etc. Different phases of task assignment to agents, like "assignment planned", "assignment requested", "assignment revoked", may be published to support situation aware worklist managers.

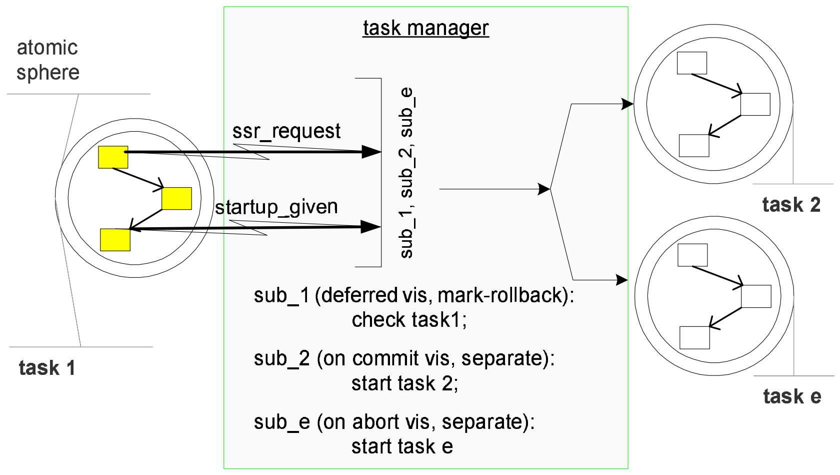

Figure 1 Event-based process enactment

Figure 1 illustrates the application of coupling modes. A task manager keeps track of the execution of business objects in order to check consistency of task executions (subscription `sub_1`); if consistency is violated, the triggering transaction will be marked rollback-only. Otherwise, the task manager chains the respective follow-up task (subscription `sub_2`), optionally using complex events to distinguish alternative execution paths depending on significant events which are external to the process. If task execution errors occur, restarts can be initiated or contingency steps may be invoked (`subscription sub_e`). As already mentioned, task managers may not only react to primitive events signalled by the controlled task but also to composed events signalled by other task managers of predecessor tasks, task managers of a different process or even external systems. Therefore X^2TS must support composition and couplings over transaction boundaries.

Events published by controlled tasks and task managers will not only be of interest in a local process enactment scope but also in a more global scope. In a recent study in cooperation with the German air traffic control authorities [30, 23], we found that the notification of regional and supra regional capacity planning systems and decision support systems about progress and critical situations in the flight departure (gate-to-runway) process is essential. In addition, worklist managers at the controller's workplace must be situation aware, that is track the status of different but inter-related process instances, and track planned, past, and current task assignments for several agents involved in inter-related processes, and timely notify the controller of significant situations.

Different event-subscribers will have different requirements with respect to visibility and coupling of atomicity spheres. For example, worklist managers will typically require immediate visible notifications and additionally react to the abort of actions or change in assignment status in separate transactions. Reliable task control flow can be implemented using *on commit/abort* visibilities whereas consistency checks typically are *deferred* and share the same transaction context. The open nature of the envisaged event-based architecture may provide the necessary flexibility to integrate next-generation air traffic control IT systems.

Activities are mapped to transactions defining atomicity spheres on the workflow schema level. In the simple case, an activity is mapped to a transaction and subactivities are mapped to (possibly open) subtransactions. Additionally, spheres of atomicity could span more than one activity. This approach shares many ideas with [13] and [1]. X^2TS does not implement multitransactions (i.e. multilevel transactions [52], nested SAGAS [19], DOM [6]) by itself, but supports the construction of multitransactions using X^2TS, as the necessary commit/abort dependencies can be established and for example compensations can be triggered by X^2TS. Recently, based on the work in [42], a proposal for *Additional Structuring of the OTS* [24] has been submitted to the OMG, which provides activity management, e.g. allows for the realization of SAGAs and compensations, on top of OTS and which we think could benefit from X^2TS. We note, that multitransactions alone do not solve the problem of workflow recovery, as multitransaction concepts all bear the problem of specifying compensations that preserve correctness in terms of serializability. Such correctness criteria typically are based on commutativity of operations [52, 26] and thus require application level knowledge. We think, that in a CORBA world it will be hard to define which operations commute and which do not. However, compensating activities at the workflow schema level seem to be a promising approach. In that case, not all local transactions of a task execution need to be compensated automatically.

4 X^2TS prototype

First, we will summarize the key features of CORBA OTS and NOS as far as relevant to the X^2TS prototype and give a short overview of which part of the services are supported by our prototype and the interfaces that are essential for using it. The design and implementation of the X^2TS prototype will be discussed later on.

4.1 CORBA OTS

OTS can be thought of as a framework to manage transactional contexts and orchestrate the two-phase-commit processing (2PC) between potentially remote recoverable servers. A CORBA transaction context - identified by its `Control` object reference - thus represents a sphere of atomicity. OTS neither provides failure atomicity nor isolation per se but delegates the implementation of recovery and isolation to the participating recoverable servers. Isolation can be either implemented by the transactional object itself or by use of the Concurrency Service [39].

A CORBA recoverable server object must agree upon a convention of registering callback objects with the OTS *Coordinator*. The latter drives the 2PC through invocation of callback methods as depicted in Figure 2 (subtransaction interfaces are left out).

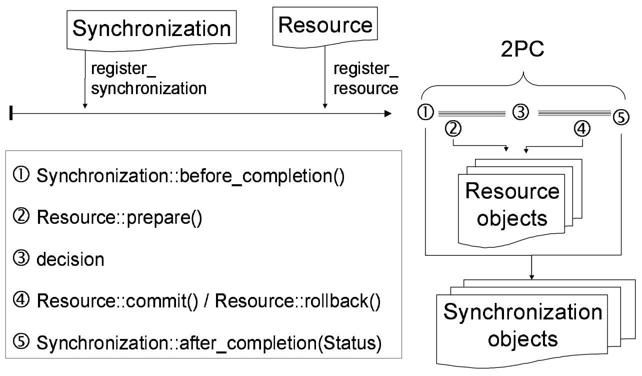
Synchronization
Resource
register_ synchronization
register_ resource
2PC
① ② ③ ④ ⑤
① Synchronization::before_completion()
② Resource::prepare()
③ decision
④ Resource::commit() / Resource::rollback()
⑤ Synchronization::after_completion(Status)
Resource objects
Synchronization objects

Figure 2 Callbacks in OTS

`Resource` objects participate in the voting phase of the commit processing and impact the outcome of the transaction. The `Synchronization` object is not essential for 2PC but for additional resource management, like the integration of XA compliant databases and queue managers. The OTS engine will call back the registered `Synchronization` objects before start of and after termination of the two phase commit processing. In X^{2}TS we make use of *Synchronization* objects to realize checked transaction behaviour and to add publishing of transaction state change events (also see section 4.4).

4.2 CORBA NOS

The CORBA Notification Service has been proposed as extension to the CORBA Event Service. The most notably improvements in NOS are the introduction of filters and the configuration of quality of service properties. NOS principally supports interfaces for push-based and pull-based suppliers and consumers. Events may be typed, untyped or structured and signalled to the consumer one-at-a-time or batch-processed. A `StructuredEvent` consists of a fixed event header, optional header fields, a filterable event body and additional payload.

An `EventChannel` is an abstraction of a message bus: all subscribers connected to a channel can in principle see all events published to that channel. Consumers may register conjunctions or disjunctions of filters, each of which matches events to a given constraint expression. Event composition is not supported by NOS - besides underspecified conjunction and disjunction operators. Quality of service (QoS) and filters can be configured at three levels providing grouping of consumers. NOS implementations may support specific QoS parameters, e.g. event ordering, reliability, persistence, lifetime etc. Configurations are programmed by setting the respective properties, represented as tagged value tuples.

4.3 Services provided by the X^{2}TS prototype

X^{2}TS integrates a push/push CORBA Notification Service and a CORBA Transaction Service. The two basic mechanisms provided by the OTS, context management/propagation and callback handling are a suitable basis for incorporating extended transaction coordination. X^{2}TS supports indirect context management, implicit context propagation and interposition. We have implemented an XA-adapter [48], mainly to integrate RDBMSs, currently there is support for accessing Informix IUS using embedded SQL in a CORBA recoverable server.

In our prototype, we only support the push-based interfaces with `StructuredEvent` one-at-a-time notifications. We assume that events are instances of some defined type and that subscription may refer to events of specific types and to patterns of events. Patterns may range from simple filters to complex event compositors. To this end, we do not specify the event model and specific types of events, but suppose the `StructuredEvent` to act as a container for whatever structured information an event of some type carries.

The overall architecture of the combined transaction and notification service as provided by X^{2}TS is shown in Figure 3.

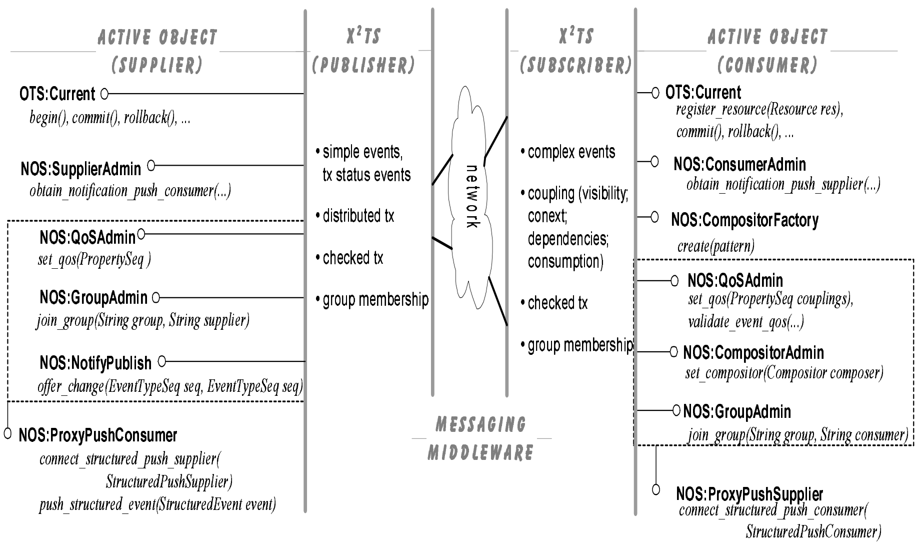
ACTIVE OBJECT (SUPPLIER)
X²TS (PUBLISHER)
X²TS (SUBSCRIBER)
ACTIVE OBJECT (CONSUMER)
OTS:Current
begin(), commit(), rollback(), ...
NOS:SupplierAdmin
obtain_notification_push_consumer(...)
NOS:QoSAdmin
set_qos(PropertySeq)
NOS:GroupAdmin
join_group(String group, String supplier)
NOS:NotifyPublish
offer_change(EventTypeSeq seq, EventTypeSeq seq)
NOS:ProxyPushConsumer
connect_structured_push_supplier(StructuredPushSupplier)
push_structured_event(StructuredEvent event)
• simple events, tx status events
• distributed tx
• checked tx
• group membership
network
MESSAGING MIDDLEWARE
• complex events
• coupling (visibility; conext; dependencies; consumption)
• checked tx
• group membership
OTS:Current
register_resource(Resource res), commit(), rollback(), ...
NOS:ConsumerAdmin
obtain_notification_push_supplier(...)
NOS:CompositorFactory
create(pattern)
NOS:QoSAdmin
set_qos(PropertySeq couplings), validate_event_qos(...)
NOS:CompositorAdmin
set_compositor(Compositor composer)
NOS:GroupAdmin
join_group(String group, String consumer)
NOS:ProxyPushSupplier
connect_structured_push_consumer(StructuredPushConsumer)

Figure 3 X^{2}TS architecture

An event supplier creates, commits or aborts transactions through the use of the `Current` pseudo object (indirect context management). Events are published on behalf of the current transaction context using the `ProxyPushConsumer` interface. The proxy is provided by the `SupplierAdmin` factory. An event supplier must register (advertise) the types of events to be published using the `NotifyPublish` interface. The supplier then publishes events by invoking the `push_structured_event` with the appropriate parameter i.e. the event type is set in accordance with the previous type advertisements.

A consumer - representing an active object - subscribes to events or patterns of events by registering a `PushConsumer` callback object with a `ProxyPushSupplier` provided by the `ConsumerAdmin` factory. Subscription to patterns of events and transactional couplings are imposed by configuring the service proxy, i.e. setting the appropriate properties through `QoSAdmin` and `CompositorAdmin`. Detecting patterns of events is realized by pluggable event compositors that are created by a specific `CompositorFactory`. We do not support the standard NOS filters but added our own proprietary `Compositor` interface.

The `GroupAdmin` interface provides group services, which are essential to realize checked transactions and vital reactions. The semantic imposed by a *vital* reaction of subscriber in a separate transaction as well as reactions in a shared context require, that the subscriber is known to the publisher site X^{2}TS before start of commit processing. Otherwise, communication failures could hinder the vital (or shared context) subscriber to be registered as a participant in the commit processing, and falsify the commit decision.

X^{2}TS currently supports service configuration only at the `ProxyPushSupplier` (`ProxyPushConsumer`) level. Event patterns are coded as strings and set in service properties at the proxy level. An event pattern contains the event type declarations and the composite event expression. If any coupling modes are specified, the couplings must refer to event types of the pattern declaration. We propose that facades should be defined which simplify configuration by providing predefined coupling mode settings. For event composition to cope with the lack of global time and network delays we introduce (in)accuracy intervals for timestamping events and suggest to use supplier heartbeats. The details are out of the scope of this paper - basic ideas can be found in [31]. We are implementing a basic operator framework for building application specific compositors, incorporating some ideas of [55].

We remark that a*ctive objects* are rather a concept than a programming language entity. It compares to the virtual nature of a CORBA object. To refer to active objects declaratively, they must be part of the CORBA object model. We think, that the specification of an active object should take place at the component level. In fact, the upcoming CORBA Components model [38] includes a simple facet for containers to provide event services and components to advertise publishers and register subscribers. Different knowledge models for specifying reactions, e.g. rules, are possible. Active objects, once specified, could use the services offered by X^{2}TS.

4.4 X^{2}TS prototype: architecture and implementation

In the next section, we will briefly describe the relevant features of the MOM used for transport of notifications and then present the architecture and some implementation details of the X^{2}TS prototype.

4.4.1 Pub/Sub middleware

X^{2}TS is implemented on top of a multicast enabled messaging middleware, TIB/ObjectBus and TIB/Rendezvous [51]. TIB/Rendezvous is based upon the notion of the *In-*

formation Bus [36] (interchangeable with “message bus” in the following) and realizes the concept of *subject based addressing*. The subject name space is hierarchical and subscribers may register using subject name patterns. Three quality of service levels are supported by TIB/Rendezvous: reliable, certified and transactional. In all modes, messages are delivered in FIFO order with respect to a specific publisher. There is no total ordering in case of multiple publishers on the same subject. Reliable delivery uses receiver-side NACKs and a sender-side in-memory ledger. With certified delivery, a subscriber may register with the publisher for a *certified session* or the publisher preregisters dedicated subscribers. Atomic message delivery is not provided. The TIB/Rendezvous library uses a persistent ledger in order to provide certified delivery. Messages may be discarded from the persistent ledger as soon as all subscribers have explicitly acknowledged the receipt. Acknowledgements may be automatic or application controlled. In both variants, reliable and certified, the retransmission of messages is receiver-initiated by sending NACKs. Transactional publish/subscribe realizes message integrating transactions [49] and is therefore not suitable for the variety of couplings we aim to support.

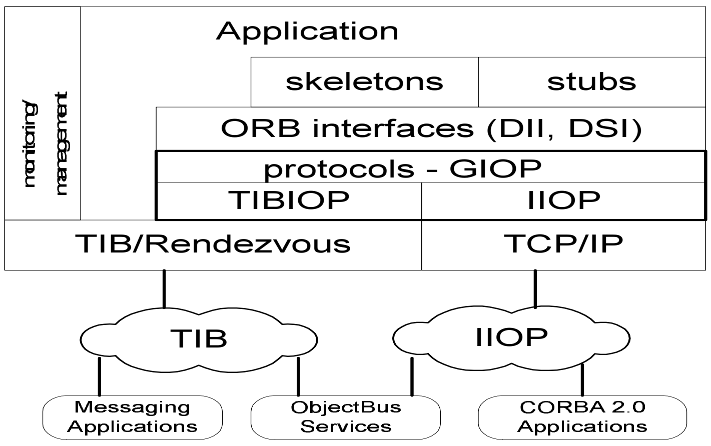

Figure 4 ObjectBus architecture

The diagram in Figure 4 depicts, how the multicast messaging middleware is introduced to CORBA in ObjectBus. The General Inter-ORB Protocol is implemented both by a standard IIOP layer and a TIBIOP layer. When using TIBIOP, the GIOP request messages are marshalled into TIB/Rendezvous messages and published on the *message bus*. In order to preserve interoperability, server objects may be registered with both, TIBIOP and IIOP profiles at the same time. Most important for the X^2TS prototype is the integration of the ORB event loop with TIB/Rendezvous, in order to be able to use the messaging API directly.

One of the design goals is to leverage the features of the underlying MOM as much as possible in order to provide asynchronous publication, reliable distribution of events and replay in case of failures. If a consumer crashes before the events have been consumed or a network error occurs, the MOM layer should initiate replays of the missed events on recovery of the consumer.

We have to ensure exactly-once delivery guarantees, end-to-end with respect to CORBA supplier and consumer. Event composition requires the possibility to consume events individually. Once an event is consumed, it must not be replayed. Additionally, consumption of events may depend on the consumer's transaction to commit. Therefore, at the consumer side, X^{2}TS supports explicit acknowledgement of received and consumed events as well as transactional acknowledgement and consumption of events. One of the complexities when realizing the prototype was to map delivery guarantees at the X^{2}TS supplier and consumer level to acknowledgements at the reliable multicast layer provided by the TIB/Rendezvous MOM. We use certified delivery and explicit acknowledgements of messages at the MOM layer, i.e. between X^{2}TS publisher and subscriber components. The publisher's ledger will persistently log events as long as there are subscribers that have not been delivered the message. In order to be able to consume events individually, the `MOM Connector` at the consumer site must persistently log consumption of events. In case of transactional subscribe, consumption of events will be logged on behalf of the consuming transaction and the consumer's transaction outcome depends on the logging to be successful. Additionally, we must filter out duplicate events, in case the event was consumed at the X^{2}TS level but not acknowledged at the MOM level.

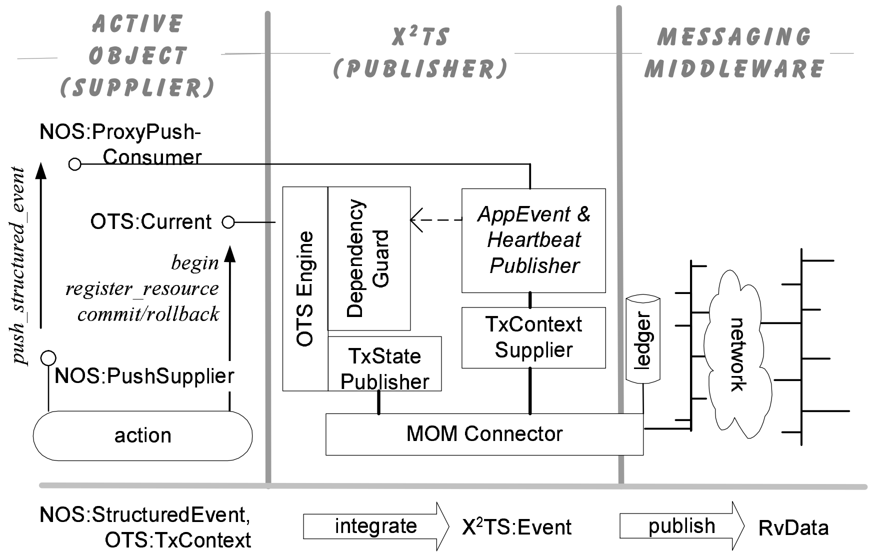

Figure 5 X^{2}TS publisher components

4.4.2 X2TS publisher components

Figure 5 depicts the components of X^{2}TS at the supplier site. The supplier has access to its transaction context through the *`Current`* interface and may push an event to the

`AppEvent Publisher`, which implements the *`StructuredProxyPushConsumer`* interface. In addition to the application events, which are signalled non-periodically, the event stream is enhanced with periodic heartbeat events. Thereby, event composition at the consumer site can correlate events based on occurrence order, instead of an unpredictable delivery order [31].

The `TxContext Supplier` is responsible for enclosing the current transaction context with the event. We use the *`OptionalHeaderFields`* of the event to piggy back transaction context information as well as event sequence numbers, and timestamps. The `MOM Connector` marshals events into a TIB/Rendezvous specific message (`RvData`) and publishes the events on the message bus.

The asynchronous nature of publishing events should be preserved, even if different consumers couple their reactions in different modes. For example, the *on commit* visibility of some consumer shall 1) not restrict other consumers to use *immediate* visibility and 2) not block the supplier in 2PC processing unnecessarily. Therefore, events are always published immediately on the message bus and visibilities and forward dependencies are resolved at the consumer site without requiring further interaction with the supplier in most cases. The design is based on the idea to publish transaction state change events asynchronously and treat visibilities and dependencies as a special case of event composition. In order to do so, the `TxState Publisher` publishes each state change of a transaction, such as 'operation-phase-terminated', 'tx-committed' and 'tx-aborted'. While such an approach would be straight forward in a centralized system, in a distributed environment with unpredictable message delays we cannot say at a consumer node, if for example a commit has not happened yet or the commit event has not been received yet. We will discuss in the section 4.6, how those situations are managed, following the principle of graceful degradation.

While forward dependencies can be resolved at the consumer site, backward dependencies impact the outcome of the triggering transaction and must be dealt with at the supplier site. The `Dependency Guard` couples the commit processing of the supplier's transaction to the outcome of reactions that are vital or that are to be executed in the same transaction context. In order to do so, for each backward coupled reaction a *Resource* object must be preregistered with the triggering transaction. Additionally, this requires to define the group of consumers which impose backward dependencies. A supplier must explicitly join this group and thereby allow the backward dependencies to be enforced.

4.4.3 X^2TS subscriber components

The components at the consumer site are depicted in Figure 6 below. The `MOM Connector` maps a X^2TS consumer's registration to appropriate MOM subscriptions. In the simple case, NOS event types and domain names are directly mapped to corresponding TIB/Rendezvous subjects. However, more complex mappings are possible and useful to leverage subject-based addressing for efficient filtering of events.

On incoming `RvData` messages, the MOM Connector unmarshals the event into an X^2TS internal representation. The components run separate threads and are connected in a pipelined manner. Events are first pushed to the `Visibility Guard`, which enforces the various visibility QoS by buffering events until the necessary condition on

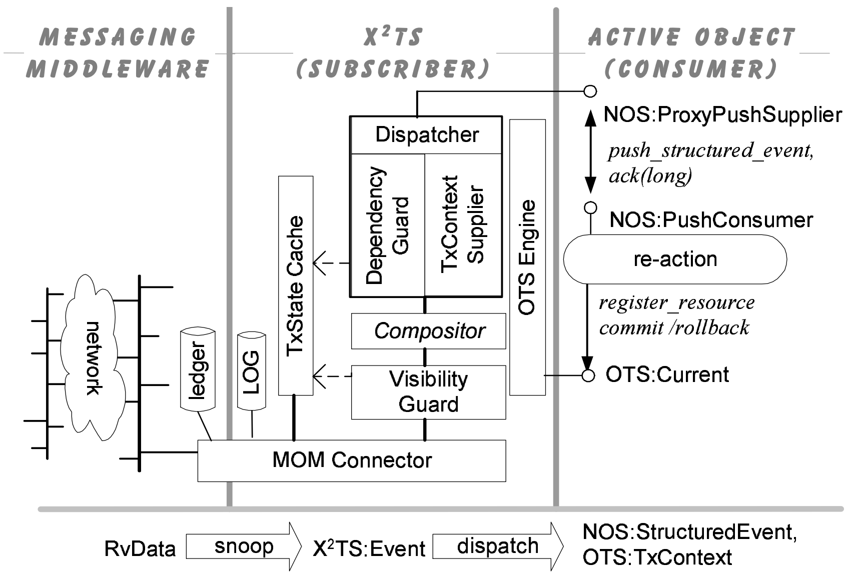

Figure 6 X^2TS subscriber components

the triggering transaction outcome is fulfilled. If events becomes visible, they are pushed to the `Compositor` (which also may be a simple filter). When the `Compositor` detects matching events, it passes them to the `Dispatcher`, which establishes the appropriate transaction context for the reaction to run in and sets up the forward and backward dependencies, before finally pushing the event to the *PushConsumer* object.

4.5 Enforcing visibilities

For each different visibility QoS which the consumer has configured, a `Visibility Guard` and its corresponding `RV Connector Pipe` are created. The `Visibility Guard` is running in its own thread and dequeues incoming events from the `MOM Connector` through the `RV Connector Pipe`. The `Visibility Guard` holds the events in a per transaction buffer until an appropriate Tx state change is signalled or a timeout occurs. Thereby, events (potentially of different types) are grouped by visibility. In addition, a transaction event listener snoops all transaction status events and forwards them to the interested `RV Connector Pipes` - and its associated `Visibility Guard` - and additionally to the `TxState Cache`.

Assume two transactions tx1 and tx2, both publishing event types e1 and e2. Further assume that the consumer subscribed *on commit* for an event pattern that contains e1 and e2 (whatever the pattern logic is).

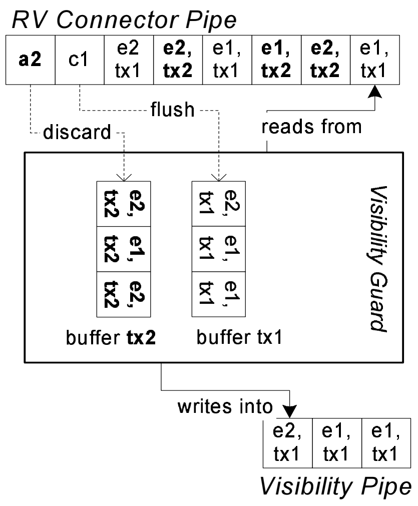

Figure 7 Visibility guard

Figure 7 depicts what the actions of the `Visibility Guard` would be in case some supplier(s) publish *e1;e1;e2* on behalf of tx1 and other supplier(s) publish *e2;e1;e2* on behalf of tx2. The buffer for tx1 would be flushed to the `Visibility Pipe` when the commit status event of tx1 arrives, the buffer for tx2 would be discarded when the abort status event of tx2 arrives. The `Compositor` - running in its own thread - would be signalled when events are pushed into the `Visibility Pipe.`

4.5.1 Transaction status propagation

`Visibility Guard` and `Dependency Guard` at the consumer site depend on the transaction status events to arrive on time. This is not necessarily so. Transaction status events are published in reliable mode, only. Therefore a commit/abort event could get lost because of communication failures. Other situations to be considered are long running triggering transactions, a supplier crash (and abort) as well as consumer crash and recovery. In all cases, the situation could arise

i) that events are buffered by the `Visibility Guard` but no transaction status event will ever trigger the application events to become visible or be discarded.

ii) completion of a triggered transaction cannot progress because of an unresolved forward dependency (see following section 4.6)

To cope with i), we additionally feed timeout events to the `Visibility Guard`. For each referenced triggering transaction, the `Visibility Guard` will then consult the local `TxState Cache` and request a current `TxStatus` event to be submitted to its `Rv Connector Pipe`. If the cache does not have a hit, it queries caches on near-by nodes. If there is no success either, then the *RecoveryCoordinator* at the supplier site will be contacted. The *RecoveryCoordinator* object is part of the OTS engine, although it is implemented as a per node daemon in a separate process. It either forwards a transaction status request to the *Coordinator*, which is colocal to the supplier. In case of a crash of the supplier process, the *RecoveryCoordinator* will inspect the transaction log for the transaction outcome. The principle of graceful degradation restricts synchronous callbacks to the supplier site to (presumably) rare occasions of major network failures. Still, eventual progress is guaranteed in spite of various failure situations.

4.6 Transaction dependencies

The `Dependency Guard` components orchestrates the behavior of dependent distributed transactions. On the subscriber site the `Dependency Guard` enforces forward dependencies, such as commit and abort dependencies by registering a *Resource* with the transaction *Coordinator*. In that case, a lightweight approach is possible to synchronize a *separate top* triggered action to the end of the triggering transaction. We may benefit from the fact that transaction status changes are pushed to the consumer site. As noted above, we cannot know at a consumer site, how long it takes the triggering transaction to complete and the commit/abort event to be delivered. In case the triggered reaction is to be completed while the status of the triggering transaction is not yet known, we heuristically decide to throw an exception and signal potentially unchecked behavior.

On the publisher site, the synchronization of distributed branches of the same transaction context and the imperative consideration of all control delegates is required. We need to consider the implications of coupling modes in case of multiple asynchronous executing reactions. We have to provide checked transaction behavior, that is we need a mechanism for joining multiple threads[1] of control for commit processing in case of coupling modes that use a shared context or specify backward dependencies. The situation may arise that the triggering transactions is to be committed while there are still branches (reactions) active. We leverage the callback framework inherent to the OTS in order to place *bolts* for checked transaction behavior and influence transaction outcome. A bolt realizes a synchronization barrier, that is checked before commit processing may start. Backward dependencies such as reactions in a shared context are realized by setting bolts at the triggering transaction. The *Synchronization* callbacks at the consumer site is then used to remove the (remote) bolts and eventually allow the triggering transaction to start commit processing. We do not block a commit call in such cases but raise an exception that the transaction is unchecked. The object in control of the transaction may then decide to retry later or take other measures. In order to establish an appropriate bolt - for example at the triggering transaction site - it is required to anticipate the reacting active objects in advance, using a group service.

5 Conclusions

X^{2}TS provides services to realize concepts of *active object systems* in a CORBA environment. X^{2}TS integrates CORBA notifications and transactions, leveraging multicast enabled MOM for scalable and reliable event dissemination. Exactly-once notifications can be realized without the need for a supplier based transactional enqueue. Notifica-

1. we refer to a thread of control in the X/Open XA sense

tions may be consumed individually and in non FIFO order by the consumer, which enables complex event composition. Coupling modes can be configured as quality of service on a per consumer basis. Therefore, the event-based application can control the manner in which to react to events published on behalf of transactions and dependencies between spheres of atomicity can be dynamically established and will be enforced by X^2TS. We consider the asynchronous nature of “loose” coupling in event-based systems and let the application decide in how far to trade time independence and flexibility of asynchronous interactions against synchronization with respect to transaction boundaries. As far as possible, visibilities and dependencies are resolved at the consumer site without calling back to the triggering application.

We suggest, that in particular process enactment can benefit from the services provided by X^2TS and multitransactions can be built on top. Still, declarative means to introduce the concept of active objects into the CORBA object model are to be explored.

Once the prototype implementation [34] is enriched with more powerful event compositors and reaches a stable state, performance experiments need to be conducted. There is still conceptual work to be done with respect to recovery of events. We suppose, that a more flexible approach to recovery of events is required than simply replaying the event history.

6 References

[1] G. Alonso, C. Hagen, H. Schek, and M. Tresch. Distributed Processing over Stand-alone Systems and Applications. In *23rd Intl. Conf. on Very Large Databases*, pages 575–579. Morgan Kaufmann, August 1997.

[2] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Massachusetts, 1987.

[3] K. Bohrer, V. Johnson, A. Nilsson, and B. Rubin. Business Process Components for Distributed Object Applications. *Communications of the ACM*, 41(6), June 1998.

[4] C. Bornovd, M. Cilia, C. Liebig, and A. Buchmann. An Infrastructure for Meta-Auctions. In *2nd Intl. Workshop on Advance Issues of E-Commerce and Web-based Information Systems (WECWIS’00)*, pages 21–30. IEEE Computer Society, June 2000.

[5] H. Branding, A. Buchmann, T. Kurdass, and J. Zimmermann. Rules in an Open System: The REACH Rule System. In *Proceedings of the International Workshop on Rules in Database Systems (RIDS ’93)*, pages 111–126, Edinburgh, Scotland, September 1993.

[6] A. Buchmann, M. Oszu, M. Hornick, D. Georgakopoulos, and F. Manola. A Transaction Model for Active Distributed Object Systems. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 123–158. Morgan Kaufmann, 1992.

[7] A.P. Buchmann. Active Object Systems. In A. Dogac, M.T. Szu, A. Biliris, and T. Sellis, editors, *Advances in Object-Oriented Database Systems*. Springer Verlag, 1994.

[8] C. Bussler. OMG Workflow Roadmap. Technical Report Version 1.2 OMG Document bom/99-08-01, Object Management Group (OMG), January 1998.

[9] A. Carzaniga, E. Di Nitto, D. Rosenblum, and A. Wolf. Issues in Supporting event-based architectural Styles. In *Proceedings of the third international workshop on Software Architecture (ISAW98)*, pages 17–20, 1998.

[10] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design of a scalable event notification service: Interface and architecture. Technical Report CU-CS-863-98, Department of Computer Science, University of Colorado, August 1998.

[11] S. Ceri, P. Grefen, and G. Sanchez. WIDE: A Distributed Architecture for Workflow Management. In *Research Issues in Data Engineering*. IEEE Computer Society, 1997.

[12] A. Chan. Transactional Publish/Subscribe: The Proactive Multicast of Database Changes. In *Intl. Conf. on Management of Data (SIGMOD 98)*. ACM Press, June 1998.

[13] Q. Chen and U. Dayal. A Transactional Nested Process Management System. In *12th Intl. Conf. on Data Engineering*. IEEE Computer Society, March 1996.

[14] C. Collet, G. Vargas-Solar, and H. Ribeiro. Toward a semantic event service for database applications. In *9th International Conference DEXA 99*, volume 1460 of *LNCS*, pages 16–27, Vienna, Austria, August 1998.

[15] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D.R. McCarthy, A Rosenthal, S.K. Sarin, M.J. Carey, M. Livny, and R. Jauhari. The HiPAC Project: Combining Active Databases and Timing Constraints. In *SIGMOD Record*, volume 17 (1), March 1988.

[16] L.G. DeMichiel, L.U. Yalcinalp, and S. Krishnan. Enterprise JavaBeans. Specification, public draft Version 2.0, Sun Microsystems, JavaSoftware, May 2000.

[17] F. Cummins. OMG Business Object Domain Task Force. White Paper bom/99-01-01, OMG, January 1999.

[18] H. Fritschi, S. Gatziu, and K. Dittrich. FRAMBOISE - an Approach to Framework-based Active Data Management Construction. In *Proceedings of the seventh on Information and Knowledge Management (CIKM 98)*, pages 364–370, Maryland, November 1998.

[19] H. Garcia-Molina, D. Gawlik, J. Klein, C. Kleissner, and K. Salem. Coordinating Multi-transaction Activities with Nested Sagas. In V. Kumar and M. Hsu, editors, *Recovery Mechanisms in Database Systems*, chapter 16, pages 466–481. Prentice Hall, 1998.

[20] R. Gruber, B. Krishnamurthy, and E. Panagos. High-Level Constructs in the READY Event Notification System. In *SIGOPS Euroean Workshop on Support for Composing Distributed Applications*, Sintra, Portugal, September 1998. SIGOPS.

[21] C. Hagen and G. Alonso. Beyond the Black Box: Event-based Inter-Process Communication in Process Support Systems. In *Intl. Conf. on Distributed Computing Systems (ICDCS)*, pages 450–457. IEEE Computer Society, 1999.

[22] M. Hapner, R. Burridge, and R. Sharma. Java Message Service. Specification Version 1.0.2, Sun Microsystems, JavaSoftware, November 1999.

[23] B. Hochberger and J. Zentgraf. Design of a workflow management system to support modelling and enactment of processes in air traffic control. Technical report, Darmstadt University of Technology, June 2000.

[24] IBM, IONA, VERTEL, and Alcatel. Additional Structuring Mechanisms for the OTS Specification. Submission orbos/2000-04-02, OMG, Famingham, MA, April 2000.

[25] G. Kappel, P. Lang, S. Rausch-Schott, and W. Retzschitzegger. Workflow Management Based on Objects, Rules, and Roles. *IEEE Bulletin of the Technical Committee on Data Engineering*, 18(1), March 1995.

[26] H.F. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *16th International Conference on Very Large Data Bases*, pages 95–106, Brisbane, Queensland, Australia, August 1990. Morgan Kaufmann.

[27] A. Koschel, S. Gatziu, G. von Bueltzingsloewen, and H. Fritschi. Unbundling Active Database Systems. In A. Dogac, T. Ozsu, and O. Ulusoy, editors, *Current Trends in Data Management Technology*, chapter 10, pages 177–195. IDEA Group Publishing, 1999.

[28] N. Krishnakumar and A.P. Sheth. Managing Heterogeneous Multi-system Tasks to Support Enterprise-Wide Operations. *Distributed and Parallel Databases*, 3(2):155–186, 1995.

[29] F. Leymann and D. Roller. Workflow-based applications. *IBM Systems Journal*, 36(1), 1997.

[30] C. Liebig, B. Boesling, and A. Buchmann. A Notification Service for Next-Generation IT Systems in Air Traffic Control. In *GI-Workshop: Multicast - Protokolle und Anwendungen*, pages 55–68, Braunschweig, Germany, May 1999.

[31] C. Liebig, M. Cilia, and A. Buchmann. Event Composition in Time-dependent Distributed Systems. In *Proceedings 4th IFCIS Conference on Cooperative Information Systems (CoopIS'99)*, pages 70–78, Edinburgh, Scotland, September 1999. IEEE Computer Press.

[32] L. Liu, C. Pu, and W. Tang. Supporting Internet Applications Beyond Browsing: Trigger Processing and Change Notification. In *5th Intl. Computer Science Conference (ICSC'99)*. Springer Verlag, December 1999.

[33] C. Ma and J. Bacon. COBEA: A CORBA-based Event Architecture. In *Conference on Object-Oriented Technologies and Systems (COOTS'98)*, pages 117–131, New Mexico, USA, April 1998. USENIX.

[34] M. Malva. Integrating CORBA Notification and Transaction Service. Master thesis, in preparation, Darmstadt University of Technology, August 2000.

[35] M. Mansouri-Samani. *Monitoring of Distributed Systems*. PhD thesis, Department of Computing, Imperial College, London, UK, December 1995.

[36] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus - An Architecture for Extensible Distributed Systems. In *SIGOPS '93*, pages 58–68, December 1993.

[37] Object Management Group (OMG). Notification service specification. Technical Report OMG Document telecom/98-06-15, OMG, Famingham, MA, May 1998.

[38] Object Management Group (OMG). Corba components (final submission). Technical Report OMG Document orbos/99-02-05, OMG, Famingham, MA, May 1999.

[39] Object Management Group (OMG). Concurrency service v1.0. Technical Report OMG Document formal/2000-06-14, OMG, Famingham, MA, May 2000.

[40] Object Management Group (OMG). Transaction service v1.1. Technical Report OMG Document formal/2000-06-28, OMG, Famingham, MA, May 2000.

[41] N. Paton, editor. *Active Rules in Database Systems*. Springer-Verlag (New York), September 1998.

[42] F. Ranno, S.K. Shrivastava, and S.M. Wheater. A system for specifying and coordinating the execution of reliable distributed applictions. In *Intl. Working Conference on Distributed Applications and Interoperable Systems (DAIS'97)*, 1997.

[43] W. Retschitzegger. Composite Event Management in TriGS - Concepts and Implementation. In *9th Intl. Conf. on Database and Expert Systems Applications (DEXA'98)*, LNCS 1460. Springer, August 1998.

[44] D.R. Rosenblum and A.L. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In *6th European Software Engineering Conference / 5th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 344–360, Zurich, Switzerland, 1997.

[45] S. Schreyjak. Synergies in a Coupled Workflow and Component-Oriented System. In *Workshop on Component-based Information Systems Engineering (CBISE '98)*, Pisa, Italy, 1998

[46] W. Schulze. Fitting the Workflow Managment Facility into the Object Management Architecture. In *3rd Workshop on Business Object Design and Implementation, OOPSLA'97*, April 1997.

[47] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Australian Unix Users Group Annual Conferece (AUUG'97)*, July 1997.

[48] R. Stuetz. Design and Implementation of a XA Adapter for CORBA-OTS. Master's thesis, Darmstadt University of Technology, October 1999.

[49] S. Tai and I. Rouvellou. Strategies for Integrating Messaging and Distributed Object Transactions. In *Proceedings IFIP/ACM International Conference on Distributed Systems Platforms and Open Distribued Processing (Middleware 2000)*, New York, USA, April 2000. Springer-Verlag.

[50] The Act-Net Consortium. The Active DBMS Manifesto: A Rulebase of ADBMS Features. *SIGMOD Record*, 25(3), September 1996.

[51] TIBCO Software Inc. TIB/ActiveEnterprise. www.tibco.com/products/enterprise.html, July 2000.

[52] G. Weikum and H.-J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 515–553. Morgan Kaufmann, 1992.

[53] S.M. Wheater, S.K. Shrivastava, and F. Ranno. A CORBA Compliant Transactional Workflow System for Internet Applications. In *IFIP Intl. Conf. in Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, September 1998.

[54] X/Open Company Ltd. Distributed Transaction Processing: The XA Specificiation Version 2 . Technical Report X/Open Document Number S423, X/Open, June 1994.

[55] D. Zimmer and R. Unland. On the Semantics of Complex Events in Active Database Management Systems. In *proceedings of the 15th International Conference on Data Engineering (ICDE'99)*, pages 392–399, Sydney, Australia, March 1999. IEEE Computer Society Press.