A Distributed Object Oriented Framework to Offer Transactional Support for Long Running Business Processes

Brian Bennett[1], Bill Hahm[2], Avraham Leff[1], Thomas Mikalsen[1], Kevin Rasmus[2], James Rayfield[1], and Isabelle Rouvellou[1]

[1] IBM Research, T. J. Watson Research Center
P. O. Box 704, Yorktown Heights, NY 10598, USA,
{bennett,avraham,tommi,jtray,isabelle}@watson.ibm.com
[2] Country Companies Insurance,
Bloomington, IL, USA

Abstract. Many business processes are both long running and transactional in nature. They are also mostly multi-user processes. Implementations such as the CORBA OTS (Object Transaction Services) modeled on the lock-based systems used for classic transactions do not fully support the requirements of such processes, and as a result, application developers must develop custom-built infrastructure – on an application-by-application basis – to support users' transactional expectations. This paper presents a novel approach to implementing long-lived transactions within distributed object environments. We propose the use of the unit-of-work (UOW) transaction model and framework, an advanced nested transaction model that enables concurrent access to shared data without locking resources. The UOW approach describes a well-structured distributed object architecture that can easily be integrated with distributed object systems. The framework offers uniform (i.e., application independent) structural transaction support for long running business processes and provides them with the semantics of traditional, short, transactions. Use of the framework enables object developers to focus on business logic, with the framework infrastructure providing functions required to support the desired semantics. We discuss the framework programming model, how it provides transactional behavior to long running business processes and some of the research challenges still ahead of us.

1 Introduction

Many business processes, such as mortgage application processing or insurance policy underwriting, can run for several days to a month or even longer. Typically, more than one person is involved in the business process. The process may start with data that is not fully validated, and that will be "cleaned up" over the course of the process; in such cases a business often does not wish to allow other processes to see the new information until it is sufficiently correct. If a customer

J. Sventek and G. Coulson (Eds.): Middleware 2000, LNCS 1795, pp. 331–348, 2000.

backs out or changes her mind, the business may want the capability to easily throw away unfinished work which it does not want cluttering its database.

The *Long Running Unit Of Work* (or LRUOW) framework provides structural transactional support for such long running business processes (LRBP). A principal contribution of the framework is that a LRBP is treated as a single long running transaction, rather than as a series of loosely connected short transactions (the approach often used to implement business processes today). Framework users interact with units of work (or UOW) that represent an application-level structure following the structure of work done at a given enterprise. Once started, a UOW may be suspended (with its state stored persistently) and subsequently resumed. It continues to exist until it completes, which it may do by committing or by rolling back. The current version of the LRUOW framework has been implemented as a set of container managed, entity, Enterprise Java Beans [1] running on top of the IBM Websphere Advanced platform [2].

Traditional transaction processing (TP) monitors such as CICS, Encina, and Tuxedo, and databases such as DB2 and Oracle have successfully abstracted an application design philosophy that separates the business logic of a flat transaction from the transactional function (ACID[1]) provided by the underlying system. However, an LRBP cannot be naively implemented on a traditional TP system because of the interaction between the following important LRBP characteristics:

- long duration (in contrast to traditional, short, transactions)
- concurrent access (in contrast to batch jobs or single-user systems)

Batch systems (in which a job is the equivalent of a long transaction) and single-user systems (such as a spread sheet application, in which the time between saves corresponds to a non-ACID transaction) do lock resources and files for moderate lengths of time (minutes to hours). Such exclusive usage is acceptable because nobody else competes for the resources. While TP monitors and databases allow concurrent usage, concurrency is provided by locking out other users when another user accesses the resource. If one application locks data for a long time, other applications that need the data must wait until the first application completes and releases its lock. Long running applications (anything over a few seconds) are thus unacceptable for a traditional transaction system.

Because an LRBP is multi-user it must be able to deal with concurrent access, and because it is long, no single user can be permitted to lock the data for the duration of the business process. Traditional TP monitors and databases,

[1] The ACID properties are
Atomicity = the transaction is either executed entirely or not executed at all
Consistency= transactions transform a persistent data store from one consistent state to another
Isolation= transactions do not read intermediate results of other non-committed transactions
Durability = once a transaction is committed, its effects are guaranteed to endure despite failures

in other words, do not fully support an LRBP's requirements, and as a result, application developers must develop custom-built infrastructure – on an application-by-application basis – to support users' transactional expectations. Such infrastructures typically intermingle business logic with transactional function. By analogy to traditional TP monitors, the goal of the LRUOW framework is to provide infrastructure that implements common transactional functions for long running business processes. By providing LRBP developers with a consistent set of transactional methods that are independent of business logic, development and maintenance effort is reduced. Note that the LRUOW framework does not intend by itself to provide full support for long running business processes, but only to provide transactional functionalities (see Section 6).

The LRUOW framework provides three major pieces of functionality to the LRBP application developer.

- packaging control of business activities into a UOW so the set of activities can be committed or rolled-back as a unit
- visibility control so that the objects created or updated are only visible within well defined scopes rather than visible to everyone
- concurrency control that manages the possibility that two users might add or change the same data in conflicting ways

The paper starts with an example of a long running business process. This example is used throughout the paper to illustrate our presentation of the LRUOW framework. The way a LRBP is divided into UOWs is explained in Section 3. Section 4 discusses how we control the visibility of the work done within the scope of a UOW. Section 5 shows how transactional behavior is provided by the framework. We conclude with a discussion on other components needed to fully support LRBPs and how they relate to our framework.

2 Example of a Long Running Business Process

This section presents a (greatly simplified) LRBP example in which an insurance company underwrites car policies. This example will be used in the next sections to illustrate the different features of the framework. The LRBP begins when a customer calls the company and requests coverage for her car. For its part, the company must create a new Policy object; it will contain relationships to new Car and Customer objects. The agent can get some information at the time of the call (Car VIN, make, model and Customer name and address), but much information can be collected only after various long running activities have completed: e.g., a credit check (Customer credit status), a car inspection (Car image), and a Department of Motor Vehicle (DMV) driver violations check. The object model and task dependency graph are shown in Figure 1. The application developers want the ACID properties that UOWs, like traditional, short running transactions, provide. Thus, in our example, the insurance company:

- wants the Policy object, comprised of the Car and Customer objects, to be created in "all or nothing" fashion (atomicity).

- wants the state of the business to move only from one valid state (valid states are defined by the business) to another (consistency).
- does not want the intermediate states of the new Car and Customer objects to be visible to other parts of the business as the information has not been validated yet (isolation).
- wants the final and intermediate results to be permanent in spite of failures (durability).

We will use this example to illustrate the LRUOW programming model and implementation in the following sections.

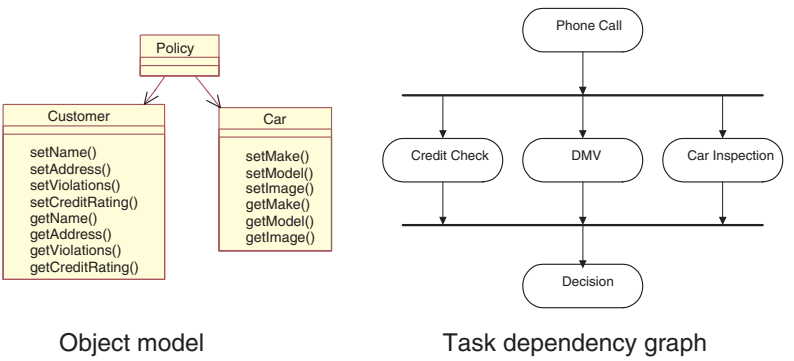

Fig. 1. Example: Object Model and Task Dependency Graph

3 Programming Model Overview

A key feature of the LRUOW programming model is that business logic is separated from the long running transaction semantics. Business object providers, in other words, concentrate on developing the function required by the long running business process: the framework is responsible for ensuring that long running transaction semantics are provided when the objects are actually deployed. In the client (application writer) view of the programming model, a LRBP contains only two types of object: the unit of work (or UOW) object which represents a nestable long running transaction (provided by the framework), and various base objects (arbitrary, non UOW-aware, objects provided by business developers). The framework takes those base objects and creates versions that are associated with the UOWs. It transparently maps method invocations under a given UOW context onto the set of objects associated with the UOW.

The LRUOW framework regards a LRBP as a directed, acyclic graph, whose nodes consist of units of work (or UOW), each of which is a nestable long running transaction [3]. Each UOW has one parent UOW (except for the root UOW

refered to as *enterprise level* UOW) and may have multiple children UOWs. The enterprise UOW owns all objects in the system and is never committed. Each sub-task of the LRBP shown in the task dependency graph of Figure 1 is mapped to a node in the uow tree (Figure 2).

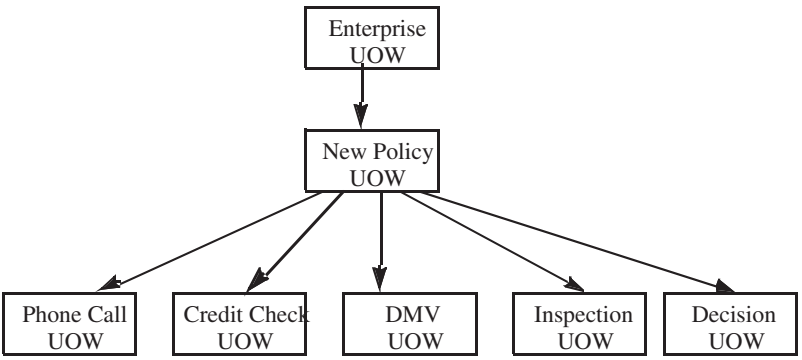

Fig. 2. UOW Tree: Note that, at a given time, only some of the leaf nodes may exist. Figure 3 and Figure 4 show snapshots of the UOW tree at different times

All activities done within the course of a LRBP are done within the context of some UOW. The UOW context is established when the client either obtains, or creates a new UOW, and joins the UOW using join() on the UOW object. Subsequent method invocations are performed within the scope of that UOW. This compares to a conventional transaction begin, or, to calling begin() on the Current object in a CORBA OTS transaction [4]. A transaction can be committed or rolled back by invoking the respective method on the UOW object. In our example, the *new policy* LRBP is initiated by requesting that the parent (or enterprise level) UOW create a child *new policy* UOW. Isolation is provided during the course of the LRBP because nodes in the UOW tree obey the following visibility rules [5]:

- The state of all objects in the scope of a parent UOW is visible to all children of that parent.
- When a child UOW commits, state changes done to all objects within the scope of the child UOW become visible to the parent UOW.
- State changes performed by a child UOW are not visible to its siblings until the child UOW commits.

An object's state and its visibility are modified over time as UOWs are created, committed, or rolled back. Continuing our example, Figure 3 represents a snapshot of the UOW tree as the *phone call* UOW (a child of the *new policy* UOW) commits. As a result, the Car and Customer objects that were created in the course of the customer's phone call become visible (and made persistent).

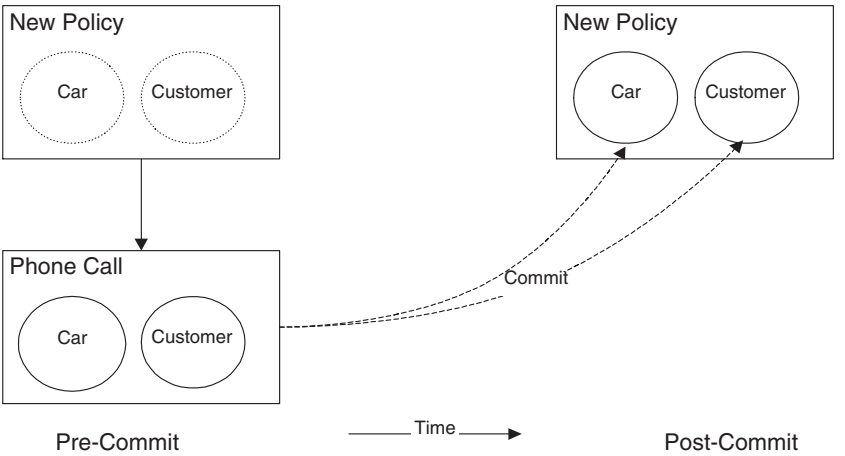

Fig. 3. UOW Tree: Effect of committing the Phone Call UOW (dotted lines are used when the objects are not visible within the scope of the given UOW)

Figure 4 is a snapshot of the LRUOW tree as one of the 2nd-level tasks (the car inspection) completes: the changes made to its version of the car (Car.image) are propagated to the parent's version. The car image was not visible to the *inspection* UOW's siblings until the commit.

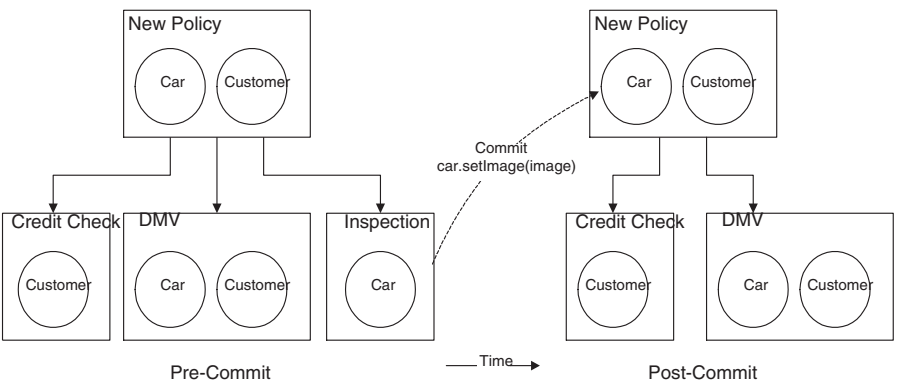

Fig. 4. UOW Tree: Effect of committing the Inspection UOW

4 Visibility and Isolation Enforcement: Facade & Version Objects

The framework uses a client/server model. In the client view of the LRUOW programming model, user interactions occur only with UOW objects and base objects; they rely on the framework to transparently map method invocations onto the set of objects associated with their UOW. The framework must enforce the protection implied by the visibility rules and, when a participant commits, must propagate the objects' state changes to the parent UOW.

The server implements this transparent mapping by ensuring that the client never actually accesses a base object instance. Instead, the client accesses a facade object that, in turn, delegates the client's method invocations to version objects that are associated with individual UOWs. The transaction context (UOW context) is implicitly propagated between the distributed EJB components that participate in the transaction, using request interceptors. UOW context propagation compares to propagation of transactional contexts in CORBA OTS transactions using implicit propagation mode.

As shown in Figure 5, each instance of a base object (e.g., a Car with VIN = 42) is associated with an instance of a facade object which wraps the set of version object instances. Each currently active UOW in which a client has referenced a facade object instance has an associated version object in the facade's version set. Through use of reflection techniques, the framework automatically generates facade and version objects from the base object. Users input the base objects as a Jar file containing, for example, Car and CarHome interfaces and CarBean and CarKey implementations. Based on this input, the framework generates the corresponding CarFacade and CarVersion EJBs, deploying them into a relational database container.

The fact that a client actually invokes methods on a facade requires the facade to extend the Car interface (as shown in Figure 5): the facade then maps from the client's UOW context (e.g., *inspection UOW*) to the corresponding Car version to which it delegates the method invocation. A Car version identity is determined by the specific Car semantics and a UOW identifier. A Car version has the Car interface, and uses the framework-independent implementation of the car (CarImpl) that is provided by the business developer (see Figure 5). The framework's task of generating the server-side facade and version objects on behalf of the client is made easier when base objects follow the *Bridge* design pattern [6]. Since clients code to an interface (e.g., Car in Figure 5), the server-side (facade) objects need only provide a shallow wrapping implementation to satisfy the contract with the client. At run-time, the server substitutes a facade for the base object.

4.1 Lifecycle

Although Figure 5 shows how the client view of a base object is actually implemented on the server by facade and version objects, it does not explain how the client gets a reference to a facade in the first place.

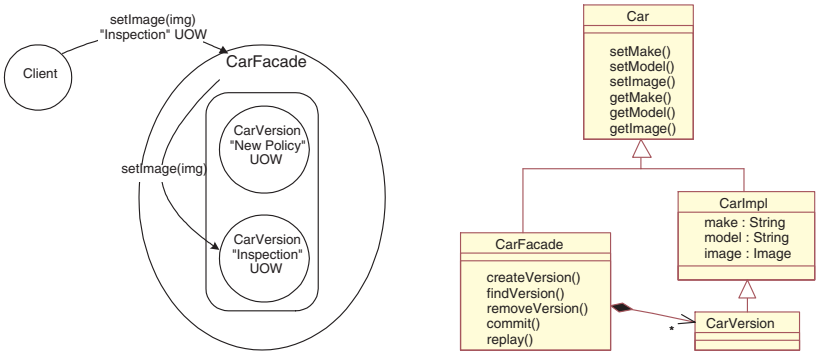

Fig. 5. Delegation of client methods / Facade and Version Car Objects

In addition to providing the base object interface and implementation (the *Car* and *CarImpl* objects), the LRUOW programming model requires the business object provider to supply an interface specifying how base object instances are created, located (queried), and removed. Our implementation follows the factory design pattern [6], so that clients access base object lifecycle function by invoking methods on the associated factory class. The LRUOW framework extends the base factory interface (e.g., with a *CarFacadeFactory*) such that the server returns facade objects to the client instead of the base object.

The challenge addressed by the framework is how the programming model used by LRBP participants – in which a single user accesses single instances of base objects – is supplied in an environment of concurrent, multi-user, access to sets of version objects. Users should be able to program as if there is only a single instance of a given object (e.g., a car with VIN = 42 where VIN is a unique key), even though the object accessed is actually one of a set of versions whose relationships are determined by the structure of the LRBP unit of work tree.

The key concept is that a business object's existence is defined relative to a specific UOW, so that an object may exist with respect to UOW_1 and not exist with respect to UOW_2. The reason for this has to do with the visibility rules discussed in Section 3 which we can now restate in terms of facade objects:

A business object exists with respect to UOW_i if and only if a non-deleted version, associated with UOW_i, exists in the facade's set of versions or the business object exists with respect to the parent of UOW_i in the LRUOW tree.

In our implementation, the facade and factory collaborate to provide business objects to clients. A client can get a reference to a facade in one of two ways: through object creation and through query (object location). Often, however, a child UOW will invoke business methods on an object whose reference was obtained in a *parent* UOW. The facade transparently creates a new version (to be associated with the child UOW) the first time that one of its business

methods are invoked. If the business object does not exist with respect to the client's UOW, the facade throws an exception. This code path (unlike the ones for creation and location through query) is managed entirely by the facade, and does not involve the factory object.

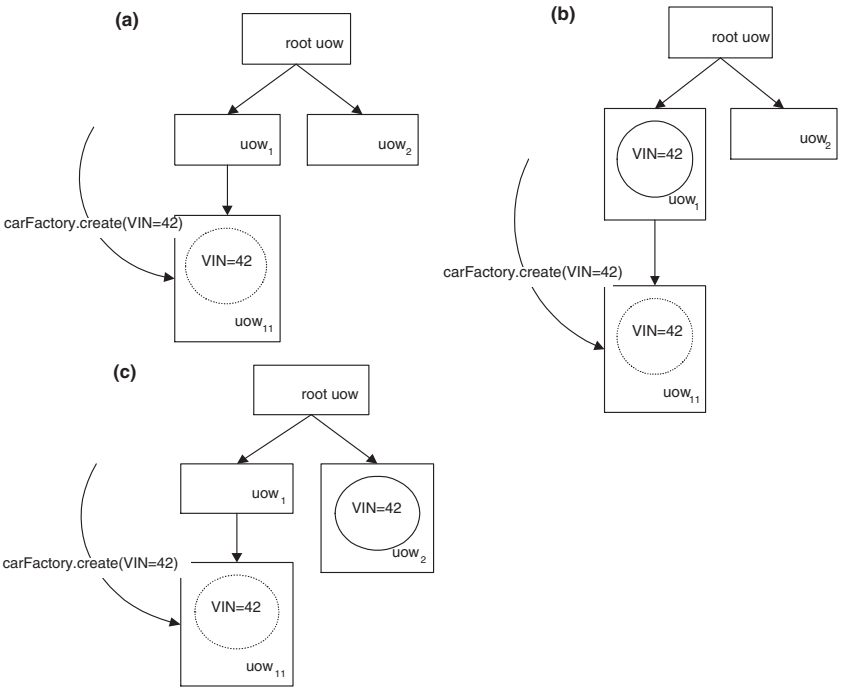

Fig. 6. (a) Client creation of a car, no pre-existing Facade car. (b) Client creation of a car, with a pre-existing version visible to client. This is an illegal state that results in an exception (c) Client creation of a car, with a pre-existing version that is not visible to the client. The resulting clash on commit is dealt with in Section 5

Creation Suppose that the base Car semantics specify that a new Car is created when a client invokes *CarFactory.create(VIN = 42)*. Figure 6 shows how the framework deals with various scenarios; note that, in these figures, the client is associated with UOW_{11}.

– No CarFacade with the specified VIN exists with respect to *any* UOW: i.e., there are no versions of a car with the specified VIN in existence (Figure 6a).
  - the CarFactory creates a FacadeCar with (VIN = 42).
  - the CarFacade creates a CarVersion that will be associated with UOW_{11}, and inserts it into the set of version objects.

- A CarFacade with the specified VIN exists with respect to UOW_{11} (Figure 6b), because a version is associated with its parent UOW_1 – even though it does not exist with respect to UOW_2.
  - the CarFactory determines that CarFacade with (VIN = 42) exists.
  - the CarFacade determines that it is visible to UOW_{11}. Since the object already exists, the facade must therefore throw a creation exception; this will be rethrown by the factory to the client.
- A CarFacade with the specified VIN does not exist with respect to UOW_{11} – even though the facade object *exists* (Figure 6c). Even though sibling UOW_2 has already created a car with the specified VIN, because UOW_2 has not yet committed (and propagated its state into UOW_0) the car does not yet exist with respect to UOW_{11}. The impending clash when the last child commits is discussed in Section 5.
  - the CarFactory determines that CarFacade with (VIN = 42) exists.
  - the CarFacade determines that it is not visible to UOW_{11}.
  - the CarFacade creates a CarVersion that will be associated with UOW_{11}, and inserts it into the set of version objects.

In order to separate a client's UOW context (which can change over the course of a LRBP) from the specific version object that is accessed at any given time, the factory returns the facade – which is responsible for mapping the client's UOW to a specific version – instead of returning the newly created version.

Location through Query Object location is the mirror image of object creation: i.e., a facade object instance can be located by a client if and only if the facade is visible to the client's UOW.

Removal A client can remove a business object if and only if it is visible to the client's UOW. Object removal is, in this sense, similar to object location. However, although the facade object cannot be actually removed until the top-level UOW commits, from the client's viewpoint once the remove method is invoked, the object no longer exists. For example, in Figure 6c, if UOW_2 deletes the CarFacade with (VIN = 42), and commits into UOW_0, the facade no longer exists with respect to UOW_1 and UOW_{11}. It is therefore valid for UOW_{11} to subsequently create a car with (VIN = 42).

To deal with such situations, whenever a client removes an object, the facade marks the associated version as *deleted* but does not remove the object from its set of versions (nor does it remove it from persistent storage). This tag allows the framework to recognize when an existing, but deleted, object actually exists with respect to a specific UOW.

5 UOW Transactional Behavior and Concurrency Management

The challenge faced in providing transactional behavior for a LRBP is that locking resources on behalf of one LRBP participant prevents other participants from

accomplishing their portion of work. On the other hand, not locking resources is unacceptable because it implies that participants cannot be given any guarantees about resource consistency. Our framework uses the following approach to provide UOW transactional behavior.

A UOW executes in two phases: a long-running phase (termed the *rehearsal*), and a short-running phase (termed the *performance*). Users accomplish work during UOW rehearsal; but, as its name suggests, no work is actually committed (in a transactional sense) during this phase. More precisely, although user work can be made persistent (so that if the system crashes, user activity will resume from the last syncpoint), the UOW does not commit and make its work visible to a parent UOW context until the user invokes UOW.commit(). If a participant instead invokes UOW.rollback(), her work will be rolled back in traditional "all or none" fashion. The purpose of the rehearsal phase is to allow long running, concurrent, activity to occur *without* locking resources – while, at the same time, the system creates a persistent copy of the information needed to resolve conflicts at commit time (performance time). Because each UOW operates on a private set of data (the versions discussed in Section 4), protection from concurrent activity is automatically provided, making lock constraints unnecessary. The performance phase is in effect a *short*, traditional transaction (with ACID properties) which modifies the versions of the objects in the parent UOW. Thus, the LRUOW framework can be implemented on top of existing transaction middleware products: the only requirement is that the system support external transactional coordination (e.g., theX/Open XA interface).

During the performance phase, the framework must deal with the concurrency issues which were ignored during the rehearsal phase. We have included two different concurrency control mechanisms. Both mechanisms seek to minimize the possibility of not being able to commit because of irreconcilable concurrent activity. Both mechanisms include the concept that not all differences between rehearsal and performance results are irreconcilable. The mechanisms vary in whether the work needs to be on the front end or on the back end, their impact on analysis and design, the types of problems they can be applied to, and the way you go about manually resolving a concurrency problem if the system can't resolve it.

5.1 Predicate & Transform Approach

As mentioned above, the framework creates during the rehearsal phase of the UOW a persistent copy of the information needed to resolve conflict at commit time. In this first approach, the information kept is the user activity or *operational log*. Facade objects record method invocations in an operational log; entries contain sufficient information to enable subsequent method replay (see Figure 7). Object state changes are preserved by logging method invocations; arguments to these methods are recorded in the log using serialization techniques. The original method invocation is later replayed using dynamic method invocation. We must deal with one subtlety: if a client invokes method1 and that method, in turn, calls method2, we log only method1 rather than logging

both methods. Replay of method1 implicitly replays method2: if method2 were to also be logged, the system would incorrectly apply method2 twice. To prevent such behavior, the system associates a *logging depth* with a given uow which is incremented each time a method is logged within that uow's scope. Log depth is decremented when the logging operation completes. An operational log record is created only if a uow's log depth corresponds to a top-level logging operation. One benefit of this approach is that facade objects are independently responsible for determining that an operation should be logged: the LRUOW framework is responsible for determining the runtime nesting of method invocations, and thus whether an individual invocation should be explicitly or implicitly logged.

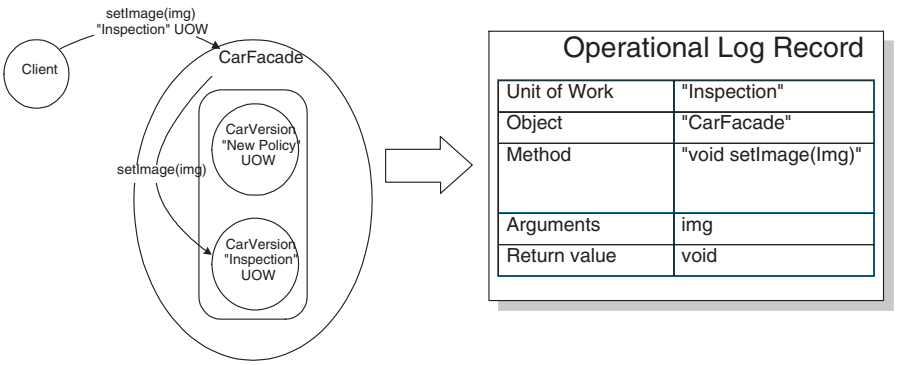

Fig. 7. Operational Log Record (there is one record per highest level transform invoked)

The operational log is a compressed copy of the LRBP in the sense that user think time, business process time, and other activities that add to the UOW's clock time (in contrast to actual business method execution time) are omitted. At performance time, the system begins a short transaction, replays the operational log, and commits the transaction to the underlying datastore if the replay is successful. Operational log replay is done with respect to a UOW's parent's data. For example, when the *inspection* UOW of Figure 4 commits, the set of methods that were invoked against its *Car* object are reinvoked against the Car version associated with the *new policy* UOW. After the UOW commits, the state of the parent UOW's versions has been updated, and reflects the state of the child UOW's versions. So, although concurrency is not an issue during rehearsal (because the *inspection* UOW manipulated its private Car version), the framework must deal with concurrency during performance since it is at this time that the *inspection* UOW Car version must be resolved with its parent *new policy* UOW Car version. Although the framework allows state changes to be applied only from within leaf UOWs, we must potentially resolve conflicts between sibling UOWs.

One approach is to log *all* methods (and their results) and allow a UOW to commit only if the rehearsal phase result matches the performance phase result. This has the advantage of being straightforward: a transaction is clearly not affected by concurrent activity if the state of all of its objects is determined only by the transaction's activity. Unfortunately, this semantic – equivalent to that of *optimistic* transactions [7] – can result in unnecessary transaction rollbacks. For example, in a debit transaction, what is important is that there are sufficient funds to cover the withdrawal – not the precise amount of funds in the account. As long as an account can cover all concurrent withdrawals, we do not want to rollback the transactions. Under the optimistic semantics, if the performance phase replays the fund withdrawals in a sequence that differs from the rehearsal phase, the difference in state (e.g., balance = getBalance()) is detected and the transaction aborted. Such behavior is especially unacceptable in the case of long running transactions: users will not be happy to be told that several weeks of work must be aborted despite the fact that the state transformations are *compatible*!

To achieve greater concurrency than what is offered by optimistic semantics, the LRUOW uses the concept of predicates and transforms. This approach is based on the concept of *field calls*. Field calls are a mechanism for increasing concurrency of short transactions by reducing the "product" of the amount of data and length of time, in which transaction locks must be held[5]. Field calls are more general (and allow more concurrency) than either *optimistic* or *timestamp* locking schemes. A field call consists of a predicate/transform pair consisting of (1) a *predicate*, which is checked at the time of the call and at commit time, and (2) a *transform*, which modifies transaction data in some way. The predicate test uses a shared-mode lock, and the lock is released as soon as the predicate is tested, thus allowing other transactions to read or update the data. If the predicate is false at the time of the field call, the transaction aborts. Otherwise, when the transaction is at phase 1 commit, it acquires exclusive locks on data involved in transforms, and the predicate is tested again. If it is false, the transaction aborts (no need to undo the transaction); otherwise the transform is applied (phase 2 of the commit), and the locks are released.

The LRUOW predicate/transform approach extends the field calls concept to long running processes. The application programmer has to code in terms of predicates and transforms (see below). Programmers must therefore be more aware of the fact that the LRUOW is running as a concurrent, transactional, program. This contrasts with the programming model of classic, short, transactions where application developers are almost completely oblivious of the transaction framework: all that is required is transaction demarcation and code to deal with situations where the transaction fails to commit. Note, however, that classic transactions often relax classic serializability semantics so as to achieve greater performance. Techniques such as cursor stability give greater concurrency at the cost of similarly forcing programmers to be aware of, and deal with the fact, that their application executes concurrently with other applications [5]. Different techniques to alleviate the strictness of serializability by allowing some degree

of inconsistency have been investigated before. One of them, epsilon serializability (ESR) allows read-only transactions that can handle a certain amount of inconsistency to exploit that property in order to increase concurrency [8,9]. Update transactions must be serializable amongst themselves. ESR works through a high-level specification of inconsistency whereas our approach allows fine-grained specifications of inconsistency that can be coded in many cases within the methods of the object itself and can be exploited by update transactions.

What is a LRUOW Transform? A transform is any state transforming method on an object. State transforming methods are replayed so as to transform the parent version to reflect the transforms applied to the child version.

What is a LRUOW Predicate? A LRUOW predicate is a piece of code checking some arbitrarily complex condition on a number of objects. It protects and validates one or several transform invocations. The simplest case of LRUOW predicates are classic predicates that perform pre-condition or post-condition validation. For example, a debit method on an account object will include a test of the account balance.

```
Account::debit(x) {
If (this.balance < x)   throw exception;
Else  this.balance -= x;}
```

The business developer will use such predicates to validate the single-pass code logic (by single pass, we refer to code execution that, in order to be valid, executes only once). This predicate will implicitly be logged as part of the debit transform and replayed during the performance phase. The replay will be successful if the current parent UOW account version has sufficient balance. So, as long as there are sufficient funds in the account, sibling UOWs can invoke the *debit* method, and still successfully commit to the parent UOW. The key idea is that a UOW will be rolled back [2] by the system only if a transform cannot be replayed (against its parent's state) because an associated predicate is no longer true. Transforms are therefore logged with the best performance achieved when the programmer specifies the least restrictive set of predicates. Concurrency will not be a problem – i.e., the replay will succeed – as long as the predicates associated with the child's transforms are not violated by the current state of the parent's version. For example, since the *phone call* UOW of Figure 3 creates new Customer and Car objects, there is no conflict between the parent's state (which does not contain these Customer and Car objects): the performance phase simply recreates the objects in the parent *new policy* UOW's scope. Of course one could imagine that there is a limit in the number of customers that an insurance company wants to consider. In this case, a predicate on createCustomer() will assert that the total number of current customers is less than the maximum

[2] Note that a non successful replay may also result in compensating actions needing to be initiated (see Section 6).

allowed. Only the business developer with knowledge of the desired semantics can specify which predicates, if any, should be associated with the transform.

In addition to regular pre/post conditions, predicates may have to be used to protect code with respect to two pass issues: i.e., issues that arise because of the way that performance-phase code derives from rehearsal-phase code and because of the manner in which state is derived during performance. For example, selectors of branch conditions that determine execution (i.e., values which control the sequencing of logged transforms) may need to be protected. The default assumption made by the framework is that differences between get results do not matter – and thus need not be replayed – since read operations cannot cause conflict among concurrent applications. But, when read-only data might affect the outcome of a UOW (e.g., the application path differs based on an attribute value, so that the commit does require that the attribute have a specified relationship to some value), the corresponding get methods can be specified as a predicate. The framework recognizes special *ApplicationPredicate* objects (APO) whose methods are always logged when invoked (like transforms). This enables the framework to supply persistence for predicates in exactly the same way that it provides persistence for transforms. An example is:

```
long  netWorth = aCustomer.getNetWorth();
//invocation of a read method
CustomerAPO.assertGetNetWorth(netWorth, aCustomer);
```

where CustomerAPO.assertGetNetWorth() is a utility transform that throws an exception if the result of invoking GetNetWorth on the customer object and netWorth are not equal in value. During rehearsal, the value of netWorth is (trivially) equal to that of getNetWorth since netWorth was just derived from that method. During performance, this transform will be replayed among all the other transforms, and the stored (rehearsal) value of netWorth compared against the current value of getNetWorth. If the values do not match, the uow predicate detects that the rule is violated, and an exception is thrown. Note that the framework will generate a generic FooApplicationPredicate object for every class Foo provided by the application writer. In particular, for every non-void method x() of Foo, the generic FooApplicationPredicate has a corresponding method assertX() to assert that invoking x() on Foo returns the same value at rehearsal and performance time (as illustrated by the customer example above). Less generic predicates will be coded by the application developer.

5.2 Conflict Detection/Resolution Approach

The second concurrency mechanism the framework offers is *conflict detection and resolution* (CD/R). The information kept by the framework in this case is not an operational log, but snapshots of the objects as they are first versioned in the child UOW (see Section 4). Upon commit, a process goes through and checks to see if any data in the parent UOW has changed since the object was copied to the child UOW. If no changes are found, the parent versions are updated to

reflect the data in the leaf UOW. If any changes are found, a *conflict* has been detected. A *conflict manager* is then invoked. The conflict manager is passed a list of participants in conflict. Its job is to select the *resolution manager(s)* that will be charged with resolving the conflict. As resolution managers are invoked, they apply the business logic necessary to resolve conflicts and arrive at the desired parent data state (in a similar way to what the manual procedures accomplish in a single-threaded multi-user system). Conflict and resolution managers have to implement an interface defined by the framework. A given uow is associated to one conflict manager and one or more resolution managers.

6 Further Challenges

The LRUOW framework provides structural transactional support for long running business processes. There are other aspects/issues however with LRBPs. Workflow systems [10] for example are concerned with the routing and sequencing of work among individuals and groups. The workflow system assists in defining resources, assigning resources, or initiating tasks. It acts as the controller of the overall business process. It takes a business process and breaks it into tasks (nodes in a process network), and defines a list of persons or programs that can perform tasks. Workflow systems manage recovery of the state of the workflow by reliably knowing which tasks have started and which have completed. They do not address recovery of resources manipulated in workflow tasks, nor provide an approach for handling contention when different tasks concurrently access shared data. The LRUOW framework addresses precisely these issues by providing transactional properties (including concurrency and durability) as well as an application model that is familiar to developers. This functionality can be programmed into workflow, but generally requires considerable custom work with high associated development and maintenance costs. Currently, the difficulty in custom coding of transactional constructs often leads developers to change natural workflow task definition or task relationships. For instance, because locking and visibility are complicated issues, a process – which may actually contain much parallel activity – will be serialized so as to sidestep the problem. Or, to avoid making "in process" data visible to other processes, much data will be inserted into separate containers instead of writing to a common database. The LRUOW framework greatly assists with such transactional concerns. Use of the LRUOW framework within a workflow can thus lead to a simplified workflow network, smaller workflow containers, a greater degree of parallelism, and less custom work.

Another issue with long running business processes is that some of the actions performed in the context of such processes (e.g., sending a letter to a customer) cannot be rolled-back. They can at best be *compensated* (e.g., sending a second letter to the customer asking to ignore the first one). The concept of compensation has been widely used in another approach to the problem of long running activities referred to as sagas [11]. A saga consists of a sequence of subtransactions T_1,..., T_n and a corresponding sequence of compensation trans-

actions C_1,..., C_{n-1} such that if the desired full sequence T_1,...,T_n fails in T_i then, by aborting T_i and executing C_{i-1},..., C_1, all trace of the overall transaction is removed. Like the LRUOW approach, sagas do not hold long term locks on data; unlike the LRUOW approach, sagas do not enforce visibility rules with the result that other transactions see intermediate results of any subsequence of T_1,..., T_n. Compensating transactions are a convenient and easily understood way of backing out transactions in simple systems. But they often need to be hand-coded, which makes it impractical to deploy sagas in large, complex, business systems. Also since humans must occasionally participate in the compensation process, the recovery process cannot be fully automated. The LRUOW framework by restricting the visibility of the work done in the course of the LRBP, and allowing to rollback (in case of failure) many of the actions performed, simplify the design of the compensation scheme.

Both the integration with workflow system and compensating schemes are currently being investigated.

Another area of investigation are strategies to integrate *legacy systems*. In order to provide ACID semantics to long running business processes, the framework makes a basic assumption: namely, that (during the performance phase) all resources used in the LRUOW can be externally coordinated to run in a single, short-running, classic transaction. A large set of systems meet this requirement – e.g., those that support the X/Open XA interface. However, many legacy datastores cannot be coordinated externally, and may not even supply transactions internally. One challenge that we must address, therefore, is whether precise semantics can be assigned to long running business processes that run on such systems.

References

1. Enterprise JavaBean Specification Version 1.0, http://java.sun.com/products/ejb/docs.html. 332
2. IBM Websphere Application Server, http://www.software.ibm.com/webservers/appserv/. 332
3. J. E. B. Moss. : Nested Transactions: An Approach to Reliable Distributed Computing. MIT Press. (1985). 334
4. Transaction Service Specification, in CORBA Services: Common Object Services Specification. http://www.omg.org. 335
5. J. Gray and A. Reuter. Transaction Processing Concepts and Techniques. Morgan Kaufmann, 1993. 335, 343
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Pattern, Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994. 337, 338
7. H. T. Kung and J.T. Robinson, On Optimistic Methods for Concurrency Control, ACM Trans. on Database Sys., Vol 6., No. 2, June 1981, pp. 213-226. 343
8. K. Wu, P.S Yu, and C. Pu, Divergence Control Algorithms for Epsilon Serializability, IEEE Transactions on Knowledge and Data Engineering, Vol. 9, No. 2, March-April 1997, pp. 262-274. 344
9. C. Pu et al., Divergence Control for Distributed Database Systems, Distributed and Parallel Databases, Vol. 3, No. 1, Jan. 1995, pp. 85-109. 344

10. Workflow Management Coalition (WfMC), http://www.aiim.org/wfmc/mainframe.htm. 346
11. Modeling Long-Running Activities as Nested Sagas, H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem. Data Engineering, Vol. 14, No. 1, March 1991. 346