

Advanced Transactions in Enterprise JavaBeans

Marek Prochazka

Charles University, Faculty of Mathematics and Physics,
Department of Software Engineering,
Malostranske namesti 25, 118 00 Prague 1, Czech Republic
prochazka@nenya.ms.mff.cuni.cz
<http://nenya.ms.mff.cuni.cz>

Abstract. Enterprise JavaBeans (EJB) is a new technology that aims at supporting distributed transactional component-based applications written in Java. In recent years, a lot of new advanced software applications have arisen, which have new requirements for transaction processing. Since EJB is modern concept that deals with transactions, the paper discusses the support of EJB for those requirements for advanced transactions and identifies weaknesses of transactions in EJB. The paper also proposes an extension of the current transactional concepts in EJB, which can be a remedy for some of the weaknesses identified. The extension, called Bourgogne transactions, allows a transaction to delegate bean objects to other transactions, to share bean objects with other transactions, and to establish flow control dependencies between transactions. Implementation issues together with pitfalls of the proposed extension are discussed.¹

1 Introduction

Enterprise JavaBeans (EJB) is an emerging standard for distributed component-based applications written in Java. One of the main goals of EJB is the support of electronic transactions on the Internet. In EJB, distributed flat transactions are supported with no means for supporting long-lived or cooperating transactions. The goal of the paper is to identify weaknesses of the transactions in EJB and to introduce Bourgogne transactions – an extension of the current EJB transactions. The purpose of the extension is not to introduce a new transaction model or to extend EJB with some of the existing transaction models. Instead, we would like to introduce an extension that brings new features to the EJB concept of transactions and allows developers to use a rich set of transaction models.

The paper is organized as follows. What are we missing in the concept of transactions in EJB is discussed in Section 2. An extension of EJB transactions, Bourgogne transactions, is introduced in Section 3 and details of its implementation are discussed in Section 4. An evaluation of the proposed extension is provided in Section 5, where goals

¹This work is partially supported by the PEPiTA project (the Eureka project number 2033), the Grant Agency of the Czech Republic (project number 201/99/0244), and the High Education Development Fund (project number 1938/2000).

achieved are discussed. The related work is discussed in Section 6 and our intentions to the future are described in Section 7. The paper concludes with Section 8.

2 What Are We Missing in EJB Transactions

Transactions became a widely-used technique for assuring data stability, application reliability and recoverability. They are used extensively in database systems, where all data manipulation operations are demarcated by transaction boundaries. Current databases use the flat transaction model [9], some use the nested transactions, transactions with savepoints, or other simple transaction models. With expansion of object-oriented and component-based architectures, requirements for instruments guaranteeing data consistency and durability have changed. In this section, the focus is put on the identification of the most important of the new requirements, new applications' aspects, and concepts, and their relation to Enterprise JavaBeans. First, several weaknesses that are common for the majority of existing transactional systems are identified. Second, weaknesses that are specific for EJB are identified and discussed.

2.1 Weaknesses of EJB Transactions Adopted From Other Transactional Systems

Although EJB is a relatively young standard, transactions in EJB suffer from many of the weaknesses of the existing transactional systems. In some cases, the reason comes from the fact that EJB have origin in classical transactional concepts, because EJB have to collaborate with legacy applications, databases, and transaction monitors.

Currently, the only supported transaction model in EJB is the distributed flat model. Many concurrent transactions can be executed and each of them is completely isolated from others. Transactions cannot cooperate on underlying database level or on the bean object level. The former is caused by the fact that today's databases do not support sharing resources, defining transaction-specific exceptions from operation conflict table, or other means. The latter is EJB-specific: beans cannot be shared between transactions, except for stateless session beans that do not have identity and thus are transaction-unaware. In EJB, there are no differences between read-only methods and methods that affect data stored in the underlying database. Allowing a transaction to invoke read-only methods on locked bean objects could enhance the application effectivity, since in current EJB all beans are locked until the transaction that had locked them commits or aborts. This, perhaps, comes from an assumption that the majority of enterprise beans will use a traditional database (connected by JDBC connections) as the storage of data.

Transactions are also completely isolated in terms of their lifecycle and flow control. A transaction is unable to change its behavior based on a state of another transaction. For example, it is not possible to mark a transaction abort-dependent on another transaction, such that if the second transaction aborts, the dependent transaction will also abort. If applications are mostly based on transactions, it is desirable to express bindings and dependencies between them. In several papers ([1], [7], [11], [18]), dependencies

between transactions are discussed. There is no reason for not supporting such an extension in EJB.

EJB has no support for long-living or open-ended transactions. A lot of bean objects can be involved in a long-lived transaction, which can reduce system effectivity and throughput, as there is no support for partial rollbacks, early-release locks, savepoints, Chained transactions, or other advanced transaction models, such as Sagas [8], where compensating actions take place. It should be possible to release bean objects during a long transaction execution, or to adopt less restrictive criteria for transaction isolation.

The component model, which is fundamental in EJB, is in fact a model with semantically rich operations. In special cases, EJB 2.0 draft [6] distinguishes between read and write methods: persistent fields of a bean with container-managed persistence are accessible via accessor methods – *setters* and *getters*. However, this concept is not employed for increasing the level of sharing bean objects; EJB container is able to realize if the bean object state has changed or not and thus to optimize storing bean object persistent fields. In our opinion, enterprise beans' methods should be treated as semantically rich operations. For each bean, a method-commutativity table that makes it possible to mark some methods as non-conflicting should be created. This would increase the application's knowledge about a bean, and, consequently, the potential for sharing a particular bean object.

2.2 EJB Specific Weaknesses

Beyond the features partially inherited from the current software applications supporting transactions, EJB have several new features that bring new challenges and are also one of the sources of EJB weaknesses.

There is no explicit locking in EJB. A transaction is not able to acquire locks on selected beans; instead, beans are implicitly locked when they are involved to a transaction.² Therefore, locking is provided on a coarse granularity level, and transactions cannot manage their locks according to the applications' requirements.

It would be helpful to allow associating selected methods with a user-defined transaction attribute. This can be beneficial in beans, where the transaction association of a particular method depends mostly on the requirements of the client transaction and beans do not care about the transaction attribute. If a client will, for example, read an account balance and the `Account.getBalance()` method is associated with the `TX_REQUIRED` transaction attribute, the `Account` bean object will be locked by the container after the `getBalance()` method invocation although it is not necessary. If the `TX_NOT_SUPPORTED` attribute is used, the client is not able to use the `getBalance()` method in the scope of a transaction, where an atomic execution of several methods takes place. Finally, if the `getBallance()` method is associated with the `TX_SUPPORTS` transaction attribute, the client is not able not to lock the `Account` bean if the `getBallance()` method is invoked in the scope of the client transaction.

²This is not true for stateless session beans, which have no identity, their instances are assigned to a transaction from a pool of instances, and thus their awareness about transactions makes no sense.

Using the current EJB, this can be solved by providing more `getBallanceXXX()` methods, each associated with a particular transaction attribute. A better solution is to allow an association of a method with a set of transaction attributes. The client could dynamically select an attribute appropriate for the actual transaction requirements. Moreover, the set of transaction attributes that can be used to associate with a particular method could be also dynamic. For example, a bean object originally associated with the `TX_SUPPORTS` transaction attribute could be switched to strictly transactional mode (e.g., because its state becomes a part of a reliable application) by allowing the `TX_MANDATORY` transaction attribute only.

The fact that the set of transaction attributes is fixed is very limiting. If more transaction models are supported, the set of transaction attributes has to be enlarged. If, for instance, nested transactions will be introduced to EJB in the future, one can imagine the `TX_REQUIRES_NEW_NESTED` transaction attribute with a similar semantics as the `TX_REQUIRES_NEW` attribute. If a method invoked in the scope of a client transaction is associated with that attribute, a newly created container-managed transaction is a nested transaction of the client transaction. The method of the transaction context propagation and use of container-managed transactions has to be specified more precisely.

In the EJB 1.1 specification, there are no means for distinguishing between methods changing a particular bean object state and between *read-only* methods. This is one of the reasons why the synchronization on a finer granularity level is not employed. The transaction isolation levels could be put to use for synchronization at the bean level, but specifying transaction isolation levels in the bean deployment descriptor was expunged from EJB 1.1. In the EJB 2.0 specification draft [6], bean accessor methods are denoted as getters or setters, which allows a container to recognize if a bean object state was changed or not and thus to optimize storing bean object states in a persistent store. The finest synchronization granularity can be obtained by considering beans as objects with semantically rich operations. In this case, the commutativity table would be provided for methods of particular beans. If two methods of a bean commute, they can be invoked on a single bean object in the scope of different transaction. If not, the synchronization policy should be applied.

In EJB 2.0, the Java Messaging Service (JMS, [10]) is integrated using special *message-driven beans*, different from session and entity beans. In JMS, sending a message can be a part of a transaction if the `XASession` interface is used, but the transaction context is never transferred to a message receiver. If a bean is specified as using the container-managed transaction demarcation, either the `TX_REQUIRED` or the `TX_NOT_SUPPORTED` transaction attributes have to be used. The specification argues that use of other transaction attributes is not meaningful for message-driven beans, because there can be no pre-existing transaction context and no client to handle exceptions. Nevertheless, it could be very useful to allow a client transaction sending transactional requests asynchronously, so the client does not have to wait for results and can obtain return values or potential exceptions later. In this case, the transaction context has to be transferred together with the message. Different JMS delivery modes and transacted sessions should be employed for enforcing transactional message passing. If

the transaction context is passed together with the message, all transaction attributes would be meaningful for message-driven beans.

The EJB specification is not clear regarding multithreaded transactions. The Java Transaction API specification (JTA, [3]) says: “The `UserTransaction.begin()` method starts a global transaction and associate the transaction with the calling thread. The transaction-to-thread association is managed transparently by the Transaction Manager. A thread’s transaction context is either null or it refers to a specific global transaction. Multiple threads may concurrently be associated with the same global transaction.” Another note says that “some transaction manager implementations allow a suspended transaction to be resumed by a different thread. This feature is not required by JTA.” Also, “Depending on the implementation of the application server, different Java threads may be involved with the same `XAResource` object. The resource context and the transaction context may be operated independent of thread context.” It is not clear, however, how a newly created thread can be associated with a previously started transaction. Most current implementations do not associate a thread with a transaction in whose scope it is started. Since synchronization is provided implicitly at the level of transactions, it is also not clear how to synchronize threads in the scope of the same transaction.

Transactions distributed to EJB servers provided by different vendors are a natural requirement. Since each vendor provides its own implementation of classes from the `javax.ejb` and `javax.transaction` packages, a programmer has to employ a private classloader for each of the involved EJB servers to isolate the proprietary implementation classes. The transaction context cannot be propagated implicitly in this case and a client has to manage the transaction context propagation to beans deployed to containers of different vendors himself.

3 Bourgogne Transactions

To address some of the weaknesses indicated in the previous section, we propose an extension of today’s EJB transaction concepts. The extension focuses on enriching EJB by new transaction primitives that allow using arbitrary transaction models.

This approach stems from ACTA [4], which provides a comprehensive formalism for specifying transaction models. We have adopted the ACTA basic idea that an arbitrary transaction model can be specified using the definition of transaction significant events (such as transaction creation, start, commit, and abort), establishing flow control dependencies between transactions, delegation of resources from a transaction to another transaction, and the definition of criteria for sharing resources between transactions.

The purpose of the paper is also to discuss implementation issues of each of the advanced primitives applied to EJB. Since the EJB specification is tightly related to the JTA specification, many of the newly proposed primitives also affect the mechanisms proposed in JTA. This paper crosses the boundary between EJB and JTA where necessary, but it is more focused on the concept of transactions in EJB and does not discuss the impact of the proposed changes to the JTA specification. The reason for this approach comes from the fact that the newly proposed API uses some primitives from

EJB (e.g., `BeanObject`), but JTA is a universal interface supporting transactions in Java generally. To design a general extension of the JTA specification could be a challenge for another paper.

The newly proposed extension to the EJB transactions is termed *Bourgogne transactions*³.

3.1 Basic Transaction Primitives: Significant Events

In Bourgogne transactions, basic transaction primitives represent the *begin*, *commit*, and *abort* transaction significant events. In contrast to ACTA, new significant events cannot be defined. More exactly, basic transaction primitives comprise methods from the `javax.transaction.UserTransaction` interface. The semantics of each of the methods can be found in the JTA specification [3].

3.2 Advanced Transaction Primitives

Advanced transaction primitives extend EJB transactions by tools for delegating resources from a transaction to another transaction, sharing resources between transactions, and establishing dependencies between transactions. The advanced transaction primitives include:

Dependencies. A transaction is able to establish a dependency on another transaction. Dependencies are conditional bindings between significant events of the participating transactions. For example, if t_j is *abort-dependent* on t_i (t_j AD t_i), then if t_i aborts then t_j also aborts.

Resource sharing. A transaction can give another transaction permissions to access data that it owns. The permission can be for reading or writing, or a transaction can permit access to parts of its data objects, e.g., by enabling to invoke only some of the objects' operations.

Delegation. A transaction can move data objects associated with it to another transaction, so that the accepting transaction becomes responsible for commit or abort of operations executed before the delegation of the objects.

To simplify the usage of the new primitives, grouping transactions and beans is introduced in our proposal. We introduce the `TransactionSet` and `BeanSet` classes that extend the `AbstractSet` class. Transactions can be added to the `TransactionSet` and beans can be added to the `BeanSet`. These newly introduced

³This name is not based on an acronym and it does not have any special meaning. The name originates from the fact that one of the first discussions on such a concept was taken in Bourgogne, France.

classes are used in the definition of the `BourgogneTransaction` interface and make it more user-friendly.

The definition of the proposed Java interfaces can be found in [17]. The new transaction primitives and their implementation in EJB are discussed in detail in the following section.

4 Implementation Issues

The basic idea is to support advanced transaction primitives by providing a new interface inheriting from the `javax.transaction.UserTransaction` interface. The full definition of the newly proposed `BourgogneTransaction` interface is shown in [17]. A client controls the lifecycle of a transaction by means of the `BourgogneTransaction` interface in similar way as the `UserTransaction` interface. In addition, he can use the new transaction primitives thanks to the `createDependency()`, `addPermission()`, and `delegate()` methods of the `BourgogneTransaction` interface. If a bean acts as a client for other beans, it can employ the `BourgogneTransaction` interface for working with advanced bean-managed transactions.

For container-managed transactions, a developer cannot directly use the newly proposed `BourgogneTransaction` interface. Instead, he has to specify the way of use of the `Bourgogne` transactions advanced primitives in the bean deployment descriptor. Transaction attributes seem insufficient for this purpose, because they indicate only several special types of container-managed transactions. More details on container-managed transactions and transaction attributes are provided in Section 4.4.

4.1 On Dependencies

Establishing dependencies between transactions allows expressing flow control relations between two or more transactions. For example, in the nested transactions, each parent is commit-dependent on all its children and, conversely, all its children are abort-dependent on the parent. The `BourgogneTransaction` interface defines methods for establishing single transaction dependencies on another transaction or a set of transactions:

```
// Create a dependency on a transaction
void createDependency(BourgogneTransaction, depType);

// Create a dependency on a set of transactions
void createDependency(TransactionSet, depType);

// Remove a dependency on a transaction
void removeDependency(BourgogneTransaction, depType);

// Remove a dependency on a set of transactions
void removeDependency(TransactionSet, depType);
```

Similarly, the `BourgogneSet` interface (presented in [17]) defines methods for establishing dependencies of a set of transactions on a single transaction or another set of transactions.

A particular dependency is canceled after it is applied. All the ACTA dependencies are based on two types of dependencies. The first dependency type specifies that an execution of some significant event (i.e., begin, commit, or abort) will force an execution of another significant event. For instance, the abort dependency is of the first type. The second dependency specifies the execution order of two significant events. The commit dependency is an example of this dependency type. Let us name the first dependency type the *enforcing* dependency and the second type the *ordering* dependency.

In EJB, the ordering dependency can be implemented by freezing an execution of the method representing a significant event. On the other hand, the implementation of the enforcing dependency depends on the fact if the commit or abort operations are involved. Basically, abort of a transaction can be enforced immediately by marking the transaction as rollback-only, while a transaction commit cannot be enforced. Thus, the dependency enforcing commit of a transaction has to be ensured by the invalidation of the dependency condition. This is applied in the strong commit dependency: if the t_j transaction is strong-commit-dependent on the t_i transaction, the $t_j.commit()$ method is invoked, and t_i is aborted, t_j has to be marked as rollback-only and aborted finally by the transaction manager.⁴ The force-commit-on-abort dependency cannot be implemented in some cases: if a transaction is aborted, then commit of the dependent transaction cannot be enforced. In the case that the dependent transaction is aborted, an exception is raised. Potential force-begin-on- dependencies form a new type of dependencies that would be able to start a transaction without the `begin()` method invocation. This is not possible in the current Bourgogne transactions, but we would like to allow this kind of dependencies in the future.

We use a set of constants that represent individual dependencies, but it is more desirable to allow employing a particular dependency by means of handling events occurred. In the future, we would like to specify dependencies by means of registering pre- and post- events of methods which represent transactions' significant events. Each dependency could be specified by the registration of methods reacting to such pre- or post- events. This model could be used for more advanced dependencies, such as dependencies on starting or finishing bean business methods, creating bean instances, or other means. This functionality could be implemented by using the Java Message Service [10].

Compensating transactions can be easily implemented by means of Bourgogne transactions in EJB. We can establish the begin-on-abort dependency between a transaction and a transaction compensating effects of the original transaction. The compensating transaction is started if the original transaction is aborted. Troubles occur if a compensating transaction aborts. This problem can be solved by establishing the force-begin-on-abort compensating transaction dependency on itself. This should be

⁴ Note that when a method representing a significant event of a transaction is frozen by the transaction manager, the transaction can be still marked as rollback-only or a new dependency which affects this transaction can be established.

combined either with a policy for a transaction timeout or with a number of allowed attempts to finish the compensating transaction.

Dependencies with container-managed transactions can be hardly established at runtime. The dependency between a container-managed transaction and a client transaction is the only dependency that makes sense if a bean method is called in the scope of the client transaction. The dependency type could be specified in the bean deployment descriptor. More details on container-managed transactions are provided in Section 4.4.

Note that establishing dependencies works correctly with bean-managed transactions. A bean-managed transaction is treated as client transaction, where a bean acts as a client for other beans.

4.2 Sharing Bean Objects

In the `BourgogneTransaction` interface, we define the following methods for giving permissions to another transaction to access bean objects:

```
// Give a transaction permission to invoke a method of
// a bean object
void addPermission(BourgogneTransaction,
                  javax.ejb.EnterpriseBean, java.lang.Method);

// Give a transaction permission to invoke any method
// of a bean object
void addPermission(BourgogneTransaction,
                  javax.ejb.EnterpriseBean);

// Give a transaction permission to invoke any method
// of a set of bean objects
void addPermission(BourgogneTransaction, BeanSet);

// Give a transaction permission to invoke any method
// of any locked bean object
void addPermission(BourgogneTransaction);
```

The `BourgogneTransaction` interface also defines methods for giving permissions to a set of transactions (these methods use the `TransactionSet` class). A transaction can give a permission to another transaction or a set of transactions to invoke a particular bean object method. In the original EJB concept, if a stateful session bean or an entity bean is used in a transaction, another transaction cannot access its methods. Our proposal allows making exceptions to this policy, so some methods of the locked bean objects can be invoked by selected transactions. Permissions can be canceled thanks to `removePermission()` methods (the full listing is in [17]).

A transaction gives a permission to another transaction to allow sharing of its bean objects. For example, if a client uses the `Account` bean object, he can give permissions to all concurrent transactions to invoke the `getBallance()` method. If the permission is given by the client transaction `t`, no conflict occurs if another transaction invokes the

`getBallance()` method. Since `t` still remains responsible for the commitment or abortment of the `Account` bean object, the code of `getBallance()` is effectively executed in the scope of `t`.

Container-managed transactions can give permissions to access bean object methods; this is allowed by extending the deployment descriptor. For example, if a method is associated with the `TX_REQUIRES_NEW` transaction attribute, it is possible to specify permissions that will be applied to all concurrent transactions. On the other hand, a transaction cannot explicitly give permission to a particular container-managed transaction. A container-managed transaction can obtain a permission from a client transaction in three ways. The client can invoke the `permit(bean, meth)` method, which gives the permission to all concurrent transactions to invoke the `meth()` method on the `bean` bean object. If the client invokes the `permit(bean)` method, he gives the permission to all concurrent transactions to invoke any method of the `bean` bean object. It is also possible to use the `permit(bean1, meth, bean2)` method, which gives the permission to container-managed transactions started during a method invocation of the `bean2` bean object to invoke the `meth()` method of the `bean1` bean object. `permit(bean1, meth1, bean2, meth2)` gives the permission to the container managed transaction started due to the `meth2()` method invocation. Note that giving permissions works correctly with bean-managed transactions.

If an X/Open XA-compliant database is employed as a persistent store through the JDBC connection, giving permission at the enterprise bean object level would lead to the write or read permission in the underlying database. However, this is not supported in today's commercial databases. Setting of the transaction isolation level is the only isolation feature that can be set up there.⁵ Isolation levels cannot be employed for giving an explicit permission to access some object (or group of objects) by another transaction.

Note that our strategy for giving permissions has to be changed, if explicit locking primitives will be introduced to EJB. This could be more practical than the current implicit locking mechanism.

4.3 Delegation of Bean Objects

The `BourgogneTransaction` interface defines methods for bean object delegation as follows:

```
// Delegate a bean object to a transaction
void delegate(BourgogneTransaction,
              javax.ejb.EnterpriseBean);

// Delegate a set of bean objects to a transaction
void delegate(BourgogneTransaction, BeanSet);

// Delegate all bean objects to a transaction
void delegate(BourgogneTransaction);
```

⁵The read-only transaction can be considered as a transaction associated with a special isolation level. For instance, Oracle uses `READ ONLY` as one of its transaction isolation levels [16].

Delegation can be seen as rewriting the computation history: if a bean object is delegated from a transaction to another transaction, the transaction manager behaves like all the operations (methods) executed by the donor transaction were executed by the acceptor transaction. For each transaction, the EJB transaction manager keeps a list of involved bean objects. If a bean object is delegated from a transaction to another transaction, the transaction manager atomically removes the bean object from the list of the donor transaction and adds it to the list of the acceptor transaction. If a group of bean objects is delegated, the same procedure is applied for each bean object from the group.

The bean object delegation cannot be provided if X/Open XA-compliant databases are employed. However, some EJB servers do not support the two-phase commit. In these EJB servers, connection pools based on the `DataSource` class are usually employed. Delegation of bean object works correctly in these EJB servers – more details are provided in [17]. If JDBC 2.0-compliant database drivers that support X/Open XA resources are employed and the EJB server supports the two-phase commit protocol (i.e., the `XADataSource` class is used and database connections are obtained using the `XADataSource.getXAConnection()` method invocation), all local transactions in a the database that work in the scope of one global EJB transaction are associated with the same transaction XID identifier. Thanks to this, if more JDBC connections to the same database are opened in the scope of the same global transaction, they can access the same data items in the database without any conflict. Delegating of bean objects from one transaction to another transaction is just a transfer of the responsibility for the commitment or abortment of operations executed in the scope of the transaction. This leads to moving parts of the database transaction log to another database transaction, which is not possible in today's commercial databases.

A client transaction can delegate its beans to a container-managed transaction by means of the `delegate(bean, delegatedBean)` or `delegate(bean, delegatedBeanSet)` methods, which delegates the `delegatedBean` bean object or the `delegatedBeanSet` set of bean objects to a container-managed transaction started on the bean bean object. If there are started more container-managed transactions (this is possible due to eventual permissions given) or no one container-managed transaction is started, an exception is thrown. The client can invoke the `delegate(bean, meth, delegatedBean)` or `delegate(bean, meth, delegatedBeanSet)` methods to specify that the bean object or the set of bean objects are delegated to the container-managed transaction started due to the `meth()` method invocation. A container-managed transaction can delegate its bean objects to the client transaction in which scope it was created. This has to be specified in the bean deployment descriptor.

4.4 On Transaction Attributes

In today's EJB, the way of the transaction context propagation and the way of employing container-managed transactions are specified by transaction attributes. In fact, transaction attributes are very limiting. The set of transaction attributes is fixed, because semantics of the transaction context propagation is defined by setting a value of a bean method's

transaction attribute. If more transaction models will be supported, the set of transaction attributes has to be enlarged. The way of the transaction context propagation and use of container-managed transactions have to be specified more precisely. Generally, the following points have to be addressed:

Transaction context propagation: A method can be executed either with the client transaction context, or the client transaction is suspended. If the client transaction is suspended, a new (container-managed) transaction can be created or the method is not invoked in the scope of any transaction. It also has to be specified, if the client is allowed to invoke a particular method in the scope of a transaction or not. In other words, a set of allowed transactional invocation patterns has to be specified.

Relations between transactions: What is necessary for specifying new transaction attributes corresponding with newly introduced transaction models is to allow using the advanced transaction primitives with the container-managed transactions. For example, the `TX_REQUIRES_NEW_NESTED` transaction attribute specifies: if a method associated with this attribute is invoked, then 1) a new transaction is always started and there are no invocation patterns that imply an exception raising, 2) if the method was invoked in the scope of a client transaction, the newly created transaction is abort-dependent on the client (parent) transaction, which is commit-dependent on the newly created (sub)transaction, and 3), at the commit time, all bean objects associated with the (sub)transaction are delegated to the client transaction.

A rich set of transaction models can be used if advanced transaction primitives are applied to the container-managed transactions. Current EJB mixes the two points indicated above and gives names to selected combinations of transaction context propagation features. In our opinion, the deployment descriptor should allow to set the way of the transaction context propagation, raising potential exceptions, creating new container-managed transactions, and setting relations between transactions by means of dependencies, giving permissions, and delegation.

To enhance container-managed transactions by the Bourgoigne transaction advanced transaction primitives, the deployment descriptor is extended as follows. In the `container-transaction` section, the `trans-attribute` value specifies the way of the transaction context propagation. Values allowed for transaction attributes remains the same as in the EJB 2.0. If the `TX_REQUIRES_NEW` transaction attribute is used, an additional information about relation between the client transaction and container-managed transaction can be specified in the deployment descriptor. For other transaction attributes, specifying additional information makes no sense. For bean-managed transactions, the Bourgoigne transactions do not allow defining relations between a client transaction and a bean-managed transaction. It cannot be specified, for instance, that a bean-managed transaction, if created, is a child of the client transaction in which scope it is created. In other words, a bean-managed transaction is always independent on the transaction in which scope it is created. This approach prevents situations, where the bean provider employs a top-level bean-managed transaction, but

since the bean-managed transaction is executed in the scope of a client transaction, its semantics can be modified by the application assembler.

For the `TX_REQUIRES_NEW` attribute, the application assembler has to specify if there are some dependencies between the client transaction and the container-managed transaction, if container managed transaction delegates its bean objects to the client transaction, and which permissions are given by the container-managed transaction in the time of its initiation. The new `client-dependency` and `cmt-dependency` elements of the deployment descriptor specify the client transaction dependency on the container-managed transaction and vice versa. These dependencies are established in the time of the container-managed transaction initiation. The `delegate` element specifies whether the container-managed transaction will delegate all its bean objects to the client transaction in which scope it was created. Other alternatives for delegation by the container-managed transaction are not possible. The application assembler can give a permission to invoke the method associated with the `TX_REQUIRES_NEW` transaction attribute by means of the `permission` element. The allowed values are `ALL`, which gives the permission to all concurrent transactions, and `CLIENT`, which gives the permission to the client transaction.

5 Evaluation

The proposed EJB transaction extension addresses some of the goals listed in Section 2. Concurrent transactions are not completely isolated; they can cooperate by means of sharing bean objects or establishing flow control dependencies between themselves. Sharing is not based on the classical read/write model; instead, it is based on the semantics of bean methods. The granularity of sharing is not set at the level of bean objects, but at the level of beans objects' methods. This is beneficial mainly for increasing the degree of sharing bean objects and thus for increasing the throughput of the EJB server. Establishing dependencies between transactions is a classical concept that allows to express the semantics of the application in terms of involved transactions.

The extension is designed so that long-running activities are supported by the introduced transactional concepts. Bean objects need not be locked during whole transactions. They can be shared with other transactions or they can be released before commit by means of delegation to another transaction, which can be committed or aborted afterward. Moreover, compensating transactions can be defined using the begin-on-abort dependency between the original transaction and the transaction that compensates effects of the original transaction.

The extension does not deal with an explicit locking mechanism. It allows to invoke methods on locked bean objects, but assumes that each bean object participating in a transaction is locked by the transaction manager. In our opinion, a transaction-aware locking service similar to the CORBA Concurrency Control Service [15] should be introduced.

The proposed transaction primitives are used for improved specification of the transaction attributes. The application assembler is allowed to specify the way of the transaction context propagation and also to specify relations between the client

transaction and a potentially created container-managed transaction by means of dependencies, giving permissions, and delegation. The set of transaction attributes is not fixed as in the current EJB; instead, an arbitrary transaction attribute can be specified. On the other hand, our extension does not allow defining “user-defined” or “dynamic” transaction attributes. The proposed extension also does not introduce a concept for asynchronous or queued transactions. This is one of our future intentions.

There is one important limitation of the proposed EJB extension: All the proposed concepts will work with the EJB platform, but some of them cannot be used if EJB applications employ traditional transactional software. Today’s databases and transaction monitors support neither transferring resources from one transaction to another nor giving explicit permissions to access locked resources. Perhaps, in the future, the concepts proposed in this paper will be employed to this kind of transactional software.

6 Related Work

Chrysanthis and Ramamrithan in [4] introduce ACTA, a formal framework for specifying extended transaction models. ACTA allows intuitive and precise specification of extended transaction models by characterizing the semantics of interactions between transactions in terms of different dependencies between transactions, and in terms of transaction’s effects on data objects. However, ACTA does not focus on the implementation of advanced transaction models in a programming language.

ASSET [1] provides a set of transaction primitives extending a programming language. Beyond the traditional transaction primitives (e.g., `begin`, `commit`, `abort`, `get_parent`) it introduces new primitives allowing creating dependencies between transactions, resource delegation, and giving permissions for an access to acquired resources. However, no implementation based on a real architecture is shown and interfaces of the proposed primitives are not defined precisely. ASSET does not use the object paradigm for the proposed primitives; it rather uses the procedural programming style.

PJama is a clone of the Java programming language that supports object persistency. In [5], the support for customizable transactions in PJama is introduced. PJama introduces the `TransactionShell` class that provides basic transaction primitives. Custom transaction models could be provided using inheritance from the `TransactionShell` class, overloading its methods, or introducing new methods representing more advanced transaction primitives. A developer of a new transaction model can use the `LockingCapability` class for the support of ignoring conflicts between transactions, delegation of responsibility for locks, and notification of a conflict detection. The way of creating an arbitrary transaction model, however, is not fairly clear. Note that PJama has no relation to the JTS or JTA specifications.

7 Future Intentions

For the future, we plan to design our idea on container-managed transactions more specifically. We would like to propose the way of specifying an arbitrary transaction attribute in the deployment descriptor. Also, “user-defined” and “dynamic” transaction attributes should be introduced. We would like to develop creating dependencies between transactions by means of registering pre- and post- events. We would like to introduce force-begin-on- dependencies that could start a transaction without the `begin()` method invocation. We also plan to employ asynchronous transactions, which should allow to transfer the context of a transaction by sending a message from a client to a bean object. For last but not least, we plan to develop a prototype of the transaction service supporting the proposed extended transactional functionality. For this purpose, we would like to take advantage of an open-source EJB server implementation.

8 Conclusion

In this paper, we introduce an extension to the concept of transactions in Enterprise JavaBeans. We identify several weaknesses of the current transactions in EJB. Our extension, called Bourgogne transactions, solves most of the weaknesses that we have identified. Bourgogne transactions allow to employ advanced transaction models to EJB. They introduce new transactional primitives allowing to establish flow control dependencies between transactions, to delegate bean objects from a transaction to another transaction, and to give permissions to access bean objects locked by a transaction. Implementation details of the proposed extension and the impact to the concept of EJB transactions are also discussed.

References

1. Biliris, A., Dar, S., Gehani, N. H., Jagadish, H. V., Ramamritham, K.: ASSET: A System for Supporting Extended Transactions. Proceedings of ACM SIGMOD International Conference on Management of Data (May, 1994)
2. Cheung, S.: Java Transaction Service 1.0 Specification. Sun Microsystems Inc. (December 1, 1999)
3. Cheung, S., Matena, V.: Java Transaction API 1.01 Specification. Sun Microsystems Inc. (April 29, 1999)
4. Chrysanthis, P. K.: ACTA, A Framework for Modeling and Reasoning about Extended Transactions Models. Ph.D. Thesis (September, 1991)
5. Daynès, L., Atkinson, M. P., Valduriez, P.: Customizable Concurrency Control for Persistent Java. In Advanced Transaction Models and Architectures, Editors: S. Jajodia, L. Kerschberg (August, 1997)
6. DeMichiel, L. G., Yalçinalp, L. Ü, Krishnan, S.: Enterprise JavaBeans Specification 2.0 Draft 2. Sun Microsystems Inc. (September 11, 2000)

7. Elmagarmid, A. K.: Database Transaction Models For Advanced Applications. Morgan Kaufmann (1992)
8. Garcia-Molina, H., Salem, K.: Sagas. In proceedings of the ACM SIGMOD Conference, 1987
9. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann (1993)
10. Hapner, M., Burrige, R., Sharma, R.: Java Message Service API Specification 1.02, Sun Microsystems Inc. (November 9, 1999)
11. Jajodia, S., Kerchsberg, L.: Advanced Transaction Models and Architectures. Kluwer (1997)
12. Matena, V., Hapner, M.: Enterprise Java Beans Specification 1.0. Sun Microsystems Inc. (March, 1998)
13. Matena, V., Hapner, M.: Enterprise Java Beans Specification 1.1 Public Release. Sun Microsystems Inc. (August 10, 1999)
14. Object Management Group: Object Transaction Service (December, 1997)
15. Object Management Group: Concurrency Control Service (December, 1997)
16. Oracle Corporation: Oracle8 Concepts, Release 8.0, Part No. A58227-01 (December, 1997)
17. Prochazka, M.: Extending Transactions in Enterprise JavaBeans. Technical Report 3/2000, Department of Software Engineering, Charles University, Prague (May, 2000)
18. Wachter, H., Reuter, A.: The ConTract Model. In Ahmed K. Elmagarmid: Database Transaction Models for Advanced Applications (1991)
19. White, S., Hapner, M.: JDBC 2.1 API. Sun Microsystems Inc. (October 5, 1999)
20. White, S., Hapner, M.: JDBC 2.0 Standard Extension API. Sun Microsystems Inc. (December 7, 1998)
21. Yang, J., Kaiser, G. E.: JPernLite: Extensible Transaction Services for WWW. CUCS-009-98, Department of Computer Science, Columbia University (1998)
22. X/Open Distributed Transaction Processing: The XA Specification (1991)