

Extending 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 component-based applications written in Java. Since distributed electronic transactions are one of the frequent applications of EJB, the paper discusses the EJB support for advanced transactional applications and proposes an extension of the current transactional concepts in EJB. The extension, Bourgogne transactions, allows a transaction to delegate bean objects to 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.

Keywords: Enterprise JavaBeans, transactions, advanced transaction models, Bourgogne transactions.

1 Introduction

Enterprise JavaBeans (EJB) is an emerging standard platform for distributed component-based applications written in Java. EJB provides not only a distribution mechanism for the communication between clients and components, it also provides services for transactions, security and persistence in a distributed environment. 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 of 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. Section 2 provides a brief overview of Enterprise JavaBeans with an emphasis to transactions. What are we missing in the concept of transactions in EJB and which are the goals for the concept extension is discussed in Section 3. An extension of EJB transactions, Bourgogne transactions, is introduced in Section 4 and its implementation is discussed in Section 5. An evaluation of the proposed extension is provided in Section 6, where goals achieved are discussed. The related work is discussed in Section 7 and our intentions to the future are described in Section 8. The paper concludes with Section 9. The full listing of new-proposed interfaces can be found in Appendix.

2 Enterprise JavaBeans

2.1 Basic concepts

The *Enterprise JavaBeans* (EJB) framework, provided by Sun Microsystems, is a component architecture intended for development of distributed, object-oriented business applications in the Java programming language. The EJB specification ([5], [6]) defines interfaces and behavior of the EJB deployment environments, *EJB servers*, and of the reusable components that execute in these environments, *enterprise beans* or *beans* for short.

An enterprise bean implements application-dependent business logic. Instances of enterprise beans are called *bean objects*. A *container* provides a deployment environment that wraps the beans during their lifecycle; every bean lives within a container. The container provides services that the contained beans can use, namely transactions, security and persistence. The EJB specification does not state the way these services are to be implemented; it only specifies the interfaces of the container through which the services are made available to the bean objects. Every bean has a *deployment descriptor*, a description of the bean's characteristics and the bean's usage of the services provided by the container.

A client accesses a bean through the *home interface* and the *remote interface*. Both interfaces are created at deployment time by special tools supplied by the EJB server provider. The remote interface reflects the functionality of the bean, also called the business methods of the bean. The home interface supports methods for creation and removal of particular bean objects, as well as methods for querying the population of the EJB objects, termed *finder methods*.

To use a business method of a bean, the client has to obtain a reference of the bean's home interface using the *Java Naming and Directory Interface* (JNDI). Using this reference, the client can create or find a bean object, and obtain a reference to a stub implementing the bean's remote interface. The stub then delegates method calls to the corresponding EJB object.

An EJB server transparently manages the population of the beans residing in the main memory. When the population of the bean objects inside a container grows beyond a certain limit, the container stores some of the not-recently-used bean objects in a secondary memory (the objects are *passivated*). Whenever a method call targets a passivated bean object, the object is brought back into the main memory by the container (the object is *activated*).

There are two types of enterprise beans: *session beans* and *entity beans*. Session beans are short-lived objects that exist on behalf of a single client and do not represent directly any shared and/or persistent data in a database. Depending upon its conversational state, a session bean can be *stateful* or *stateless*. An entity bean, on the other hand, typically represents persistent data, usually stored in a database. An entity bean is transactional, allows shared access from multiple clients, and can be long-lived.

2.2 Transactions in EJB

EJB supports distributed flat transactions. The distribution mechanism makes it possible to involve bean objects on multiple EJB servers or to update data in multiple databases in a single transaction. Every client method invocation on a bean is supervised by the bean's container, which makes it possible to manage the transactions according to the *transaction attributes* that are specified in the corresponding bean's deployment descriptor. A particular transaction attribute can be associated with an entire bean and apply to all its methods, or just with an individual method. The scope of a transaction is defined by the *transaction context* that is shared by the

participating bean objects.

Basically, a client obtains a transaction context, which is then implicitly transferred with the client's requests. What a client can do with a transaction is to start it using the `tx.begin()` method, or to terminate using the `tx.commit()` or `tx.abort()` methods. All method invocations between the transaction begin and end are associated with the transaction context. More precisely, the way of transaction context transfer depends on the transaction attribute associated with the invoked method. For example, if a method associated with the `TX_NOT_SUPPORTED` transaction attribute is invoked in the scope of a client transaction, the transaction is suspended during the method execution.

Moreover, *container-managed transactions* are introduced. A container-managed transaction is used for a method invocation if it is required by the transaction attribute associated with the method. For instance, when the `TX_REQUIRES_NEW` transaction attribute is specified, a potential client transaction is suspended during the execution of the method associated with this attribute, and the method is executed in the scope of a new transaction created by the container. If no error occurs, the transaction is committed, otherwise it is aborted. The client transaction is then resumed.

EJB also introduces *bean-managed transactions* that are explicitly managed by a bean object. If a bean is associated with the `TX_BEAN_MANAGED` transaction attribute, it can get a transaction context and start or finish a transaction. Such a bean object can act as a client for other bean objects (this, however, is also possible with container-managed transactions).

When a client commits a transaction, all effects of the transaction are stored in a persistent store: entity beans with container-managed persistence are stored by the container, `store()` methods of entity bean objects with bean-managed persistence and `beforeCompletion()` and `afterCompletion()` methods of session bean objects implementing the `SessionSynchronization` interface are invoked by the container. In addition, the container manages opened JDBC connections, associates them with executing transactions, and sends commit or abort to connected databases during a transaction termination.

Some of the features of EJB transactions are discussed in more details in the rest of the paper.

3 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 (DBMS), where all data manipulation operations are demarcated by transaction boundaries. Current databases use the flat transaction model ([15]), 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.

3.1 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, or 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 resource managers (e.g., 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 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 (e.g., by employing transaction isolation levels) 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 ([16], [17], etc.), dependencies between transactions at the level of transaction primitives are discussed. There is no reason for not supporting such an extension in EJB.

There is no support for long-living or open-ended transactions in EJB. 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 ([19]) 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 a transaction isolation.

The component model, which is fundamental in EJB, is in fact a model with semantically rich operations. It is rather simplifying to divide methods into read and write methods, as done in the EJB 2.0 draft specification ([6]). 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.

3.2 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. (Note that this 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.) Thus, locking is provided on a coarse granularity level, and transactions cannot manage their locks according to the applications' requirements.

The EJB specification is not clear regarding multithreaded transactions. The Java Transaction API specification ([10]) says: "The `UserTransaction.begin` method starts a global transaction and associates 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.

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 bean does not care about the transaction attribute. If a client would like, for example, to read an account balance, and the `Account.getBalance` method is associated with the `TX_REQUIRED` transaction attribute, the `Account` bean object will be locked although it does not have to be. 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. 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 associating 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 an "important" 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 will be 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 (note that specifying transaction isolation levels in a bean deployment descriptor was expunged from the EJB 1.1 specification). In the EJB 2.0 specification draft, bean methods are denoted as "read" or "write" to allow a container to recognize if a bean object state was changed or not. This allows the container 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.

The transaction diamond scenario is introduced in the EJB 1.1 specification: "An example (not realistic in practice) is a client program that tries to perform two purchases at two different stores within the same transaction. At each store, the program that is processing the client's purchase request debits the client's bank account." Containers should support local diamonds, where all components are deployed to the same container. On the other hand, the situation, where two clients executing in the same transaction context concurrently access an entity bean object, is considered an application error. The example depicted above is fairly tendentious; the example can be slightly modified to a distributed transaction in which context a hotel room and a flight are reserved and payed using the same bank account. Both reservations may be executed in a different place using different beans and containers. In our opinion, EJB definitely has means for handling the distributed diamond scenario correctly such that clients involved in the same global transaction would not have to access an entity bean object serially.

Asynchronous operations are not supported in EJB. It is 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 the EJB 2.0 draft, Java Messaging Service (JMS, [9]) is integrated using special "JMS beans", different from session and entity beans. In JMS, however, sending a message is a part of transaction only if the `XASession` interface is used and the transaction context is never transferred to the message receiver. It is desirable to transfer the transaction context with a message, since a message can have exactly the same effect on a bean object as a method invocation.

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

3.3 Goals for an extension of EJB transactions

In the previous paragraphs, some of the weaknesses of EJB transactions were indicated. We believe that EJB have to be extended to improve the transactional functionality and to resolve some of the weaknesses. Apart from technical requirements, there are several general requirements for the extension:

1. **Compatibility with the current EJB:** The most important requirement is to support backward compatibility with the EJB standard. It is desirable that applications that use the current EJB transaction primitives will work without limitations or any impact on their behavior or correctness.
2. **Orthogonality:** All transactional concepts introduced in EJB can be used if they are combined with the newly introduced transactional primitives. For example, transaction attributes or container-managed transactions will work correctly combined with advanced models of transactions.
3. **Extensibility:** The EJB extension should not be based on introducing new transaction models or employing several existing ones. It should rather introduce a new transactional functionality that would allow EJB applications to employ richer transactional concepts, to define new transaction correctness criteria, etc.
4. **Scalability:** The extension should be scalable to support large (e.g., Internet-based) applications, where thousands of concurrently accessing users, thousands of EJB servers, or thousands of nodes can take place.

Of course, the basic concept of transactions, based on the ACID properties (potentially slightly restricted), has to be adopted by the EJB extension. Also, the extension should not affect the distribution of transactions.

4 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 supporting big set of transaction models.

The intention of the proposal is not to come up with a new transaction model; instead, we would like to propose a solution allowing to work with an arbitrary transaction model in EJB. This approach stems from ACTA ([2]), 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 basic transaction primitives corresponding to transaction significant events, such as transaction creation, start, commit, and abort, and advanced transaction primitives that allowing delegating resources from a transaction to another transaction, sharing resources between transactions, and establishing dependencies 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 is focused on the concept of transactions in EJB and does not discuss the impact of the proposed changes to the JTA specification. The newly proposed extension to the EJB transactions is termed *Bourgogne transactions*.¹

4.1 Basic transaction primitives: significant events

In Bourgogne transactions, basic transaction primitives are those supported by the current JTA specification. The basic primitives allow transactions to be created, started, and finished. These operations are frequently called *significant events*, since they define the basic lifecycle of a transaction. Also, the status of each transaction can be examined and the transaction timeout can be set. More exactly, basic primitives comprise methods from the `javax.transaction.UserTransaction` as follows:

```
void begin();
void commit();
int getStatus();
void rollback();
void setRollbackOnly();
void setTransactionTimeout();
```

The semantics of each of the methods can be found in the JTA specification ([10]).

4.2 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 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.

Moreover, to simplify the usage of the new primitives, grouping transactions and beans is introduced in our proposal. The definition of the proposed Java interfaces can be found in Appendix. The new transaction primitives and their implementation in EJB are discussed in detail in the following section.

¹ The name *Bourgogne transactions* is not based on an acronym, it does not have any special meaning. The name originates from the fact that one of the first debates on such a concept was taken in Bourgogne, France.

5 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 `BourgogneTransaction` interface is shown in Appendix.

5.1 Grouping transactions

It is helpful to allow a programmer to work not only with particular transactions or bean objects, but also with groups of transactions and bean objects. We introduce the `TransactionSet` and `BeanSet` classes as follows:

```
class TransactionSet extends AbstractSet {
    void add (javax.transaction.UserTransaction);
}

class BeanSet extends AbstractSet {
    void add (javax.ejb.EnterpriseBean);
}
```

Transactions can be added to the `TransactionSet` and beans can be added to the `BeanSet`. These newly introduced classes are used in the definition of the `BourgogneTransaction` interface and make it more user-friendly.

5.2 On dependencies

Establishing dependencies between transactions allows to express 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, int);

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

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

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

The `BourgogneSet` interface defines methods for establishing a set of transactions dependencies on a single transaction or another set of transactions:

```
class BourgogneSet extends TransactionSet {
    void add (BourgogneTransaction);

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

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

// Remove a dependency on a transaction
```

```

    void removeDependency(BourgogneTransaction, int);

// Remove a dependency on a set of transactions
    void removeDependency(TransactionSet, int);
}

```

In the following paragraphs we show that all the dependencies used in ACTA can be handled correctly in the EJB environment. For each dependency, we present its definition and explain the implementation in EJB:

Commit dependency (t_j CD t_i): If both t_i and t_j commit then the commitment of t_i precedes the commitment of t_j . If the $t_j.commit()$ method is invoked and the t_j Bourgogne transaction is commit-dependent on another Bourgogne transaction t_i , the EJB transaction manager (TM) freezes the $t_j.commit()$ method execution until the t_i transaction commits or aborts. If the $t_j.commit()$ method is invoked and t_i has already been finished, the dependency is applied implicitly (i.e., the TM need not do anything to fulfill the dependency conditions).

Strong commit dependency (t_j SCD t_i): If t_i commits then t_j also commits. The commitment of a transaction can be hardly ensured. We have to prevent the situation where the t_i transaction commits and the t_j transaction aborts. In this case, t_i has to be aborted. More precisely, the $t_i.commit()$ method invocation has to be frozen by the TM until the t_j transaction is finished (i.e., committed or aborted). Then, if t_j is aborted, t_i is also aborted. If t_i is aborted earlier than t_j finishes, t_j is marked as rollback-only. If t_j is committed earlier than t_i finishes, the $t_i.commit()$ method invocation can be executed by the TM without any constraints. If t_i is aborted, the dependency is applied implicitly.

Abort dependency (t_j AD t_i): If t_i aborts then t_j aborts. If t_j is abort-dependent on t_i , when t_i aborts and t_j is still active, t_j is marked as rollback-only. If the $t_j.commit()$ or $t_j.abort()$ methods are invoked and t_i is still active, t_j has to be frozen until t_i finishes. Then, if t_i commits, the TM can continue the execution of the invoked $t_j.commit()$ or $t_j.abort()$ methods, and if t_i aborts, t_j is also aborted.

Weak abort dependency (t_j WD t_i): If t_i aborts and t_j has not been committed then t_j aborts. If $t_i.abort()$ is invoked (or $t_i.commit()$ is invoked and the t_i transaction has been marked as rollback-only) and t_j is still active, t_j is marked as rollback-only.

Termination dependency (t_j TD t_i): Transaction t_j cannot commit or abort until t_i either commits or aborts. The $t_j.commit()$ or $t_j.abort()$ method invocations are frozen by the TM until the $t_i.commit()$ or $t_i.abort()$ methods are invoked.

Exclusion dependency (t_j ED t_i): If t_i commits and t_j has begun executing, then t_j aborts. If the $t_i.commit()$ method is invoked, the t_j transaction is marked as rollback-only by the TM. If the t_j transaction has not started before the t_i commitment, the t_j transaction is not marked as rollback-only by the TM after the t_j transaction startup and the dependency is applied implicitly.

Force-commit-on-abort dependency (t_j CMD t_i): If t_i aborts, t_j commits. This dependency cannot be satisfactorily handled by the TM, because the commitment of a transaction can be hardly ensured. If the t_i transaction aborts as first, then the t_j transaction can still be aborted. In this case, an exception is raised.

Begin dependency (t_j BD t_i): Transaction t_j cannot begin execution until t_i has begun. The $t_j.begin()$ method invocation is frozen by the TM until the $t_i.begin()$ method is invoked.

Serial dependency (t_j SD t_i): Transaction t_j cannot begin execution until t_i either commits or aborts. The $t_j.begin()$ method invocation is frozen by the TM until the $t_i.commit()$ or $t_i.abort()$ methods are invoked.

Begin-on-commit dependency (t_j BCD t_i): Transaction t_j cannot begin until t_i commits. The `tj.begin()` method invocation is frozen by the TM until the `ti.commit()` method is invoked. If t_i is aborted, an exception is raised. If the `tj.begin()` method is invoked after the commitment of the t_i transaction, the dependency is applied implicitly.

Begin-on-abort dependency (t_j BAD t_i): Transaction t_j cannot begin until t_i aborts. The `tj.begin()` method invocation is frozen by the TM until the `ti.abort()` method is invoked. If t_i is committed, an exception is raised. If the `tj.begin()` method is invoked after the t_i transaction aborts, the dependency is applied implicitly.

Weak begin-on-commit dependency (t_j WCD t_i): If t_i commits, t_j can begin execution after t_i commits. The `tj.begin()` method invocation is frozen by the TM until the `ti.commit()` method is invoked. If t_i is aborted, the dependency is applied implicitly.

If a particular dependency is applied, it is discarded. All the dependencies described above 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 `tj.commit()` method is invoked, and t_i is aborted, t_j has to be marked as rollback-only and aborted finally by the TM². The force-commit-on-abort dependency cannot be implemented in some cases: If some 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.

One of the weaknesses of our proposal is the method of the specifying a particular dependency. We use a set of constants representing individual dependencies. It is more desirable to specify a particular dependency by expressing its semantics. This corresponds with our aim to support, preferably, every transaction model that can be implemented by means of transaction dependencies, delegation of resources, and giving permissions to access resources. 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. Probably, this functionality could be implemented by using the Java Message Service (JMS, [9]).

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

² 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.

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

Dependencies between 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 a client transaction. The dependency type could be specified in the bean deployment descriptor. Note that establishing dependencies works correctly with bean-managed transactions.

5.3 Sharing bean objects

The current EJB specification does not allow sharing bean objects between transactions. Transaction synchronization on entity beans is provided either by the container or by the underlying database. In both cases, transactions are always serialized. If a stateful session bean object's method is invoked in the context of a transaction, the bean object becomes associated with the transaction. The bean object is locked by the transaction until the transaction terminates. Stateless session beans have no state and no identity; a transaction cannot select a particular instance, since instances are grouped into pools and each of the available instances can be used for a request dispatch. Thus, transactions accessing stateless session beans are not serialized, but applications are not able to profit from this fact.

In the `BourgogneTransaction` interface, we define the following methods for giving permissions to other transactions 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);

// Give a set of transactions permission to invoke a method of a bean object
void addPermission(TransactionSet, javax.ejb.EnterpriseBean,
    java.lang.method);

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

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

// Give a set of transactions permission to invoke any method of any locked
// bean object
void addPermission(TransactionSet);
```

Permissions can be canceled thanks to `removePermission()` methods (the full listing of these methods is in Appendix). The purpose of our proposal is to allow a transaction to give a permission to another transaction to access a particular enterprise bean object or a group of bean objects. Since the EJB 2.0 specification draft introduces differences between methods affecting a bean object state (*write methods*) and methods that read

only a bean object data attributes (*read methods*), one of the possible solutions is to give a permission to read or write. For example, if we give the read permission to a transaction, then all read methods can be invoked by this transaction.

However, it is not very useful to employ the read/write model in EJB, if components are objects with semantically rich operations. Thus, we introduce a model that allows a transaction to give a permission to another transaction 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 to make exceptions to this policy, so some methods of the locked bean objects can be invoked by some transactions.

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 using the `permit(bean, meth)` method invocation, which gives the permission to all concurrent transactions to invoke the `meth` method on the `bean` bean object, or using the `permit(bean)` method invocation, which gives the permission to invoke any method of the `bean` bean object. However, it is possible to introduce the `permit(bean1, method, bean2)` method that 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. We plan to introduce such an extension in the future. Note that giving permissions works correctly with bean-managed transactions.

If a X/Open XA-compliant database is employed as a persistent store through the JDBC connection, the permission at the enterprise bean object level leads to the write or read permission in the underlying database. However, this is not supported in today's commercial databases.

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

5.4 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);
```

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, most current implementations do not support the two-phase commit, because appropriate JDBC 2.0 drivers supporting the XA resources are not available. In current implementations, connection pools based on the

`DataSource` class are usually employed. If a transaction invokes the `DataSource.getConnection()` method, either a new connection to the database is created, or the previously created connection associated with the transaction is used. Many bean objects involved in the same transaction can share the same JDBC connection. If each bean object has its own JDBC connection, it would be easy to delegate bean objects between transactions. In this scenario, there are local database transactions on each of the JDBC connections and local transactions in the EJB server. For each transaction, the TM manages lists of affected beans and opened JDBC connections. Delegating a bean object causes moving this bean object and its JDBC connection from one transaction to another transaction. Therefore, a local database transaction corresponding with the JDBC connection starts to be associated with the acceptor EJB transaction. If an EJB transaction is committed, all local database transactions associated with this transaction are also committed. This could not be provided in the two-phase-commit fashion. Transactions aiming to employ advanced transaction primitives are forbid to share JDBC connections. For example, if a transaction affects thousands of instances of the same enterprise bean, it is ineffective to create a JDBC connection for each of the objects. If no advanced transaction primitive is used, all the objects can share the same JDBC connection and thus the same local database transaction. One of these two alternatives can be selected by invoking either the `getUserTransaction()` method or the `getBourgeoisTransaction()` method.

If JDBC 2.0 database drivers supporting XA resources are employed and the EJB server supports the two-phase commit (i.e., the `XADataSource` class is used and database connections are obtained using the `XADataSource.getXAConnection()` method invocation), all database local transactions within one EJB global transaction are associated with the same transaction XID identifier. So, even more JDBC connections to the same database are opened and because they are associated with the same transaction XID, they can access the same data items in the database. Delegating bean objects with opened JDBC connections is not possible, since the underlying database would have to support moving parts of transactions' log from a local transaction to another local transaction.

Bean objects cannot be delegated to container-managed transactions. However, a container-managed transaction can delegate all its bean objects to a client transaction in which scope it was created. This can be specified in the bean deployment descriptor.

5.5 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. Generally, the following points have to be addressed:

1. **Transaction context propagation:** A method can be executed either with a 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.
2. **Relations between transactions:** What is necessary for specifying new transaction attributes corresponding with new introduced transaction models is to allow to use the advanced transaction primitives with 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 container-managed transactions.

Current EJB mixes the two points 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.

6 Evaluation

The proposed EJB transaction extension addresses some of the goals listed in Section 3. 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 that 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 TM. In our opinion, a transaction-aware locking service similar to the CORBA Concurrency Control Service [13] should be introduced.

The proposed transaction primitives can be used for improved specification of the transaction attributes. The transaction attributes are more generic if the way of the transaction context propagation is specified by means of the specification of handling the client's transaction context and the specification of relations between the client transaction and a potentially created container-managed transaction. 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 to define "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.

The extension is compatible with the current EJB so that all the current EJB transaction primitives work without limitations and correctly. The extension is not based on introducing a new set of transaction models, it introduces new transaction primitives that extend the EJB transaction concept.

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.

7 Related Work

Chrysanthos and Ramamrithan in [2] 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. During last years, ACTA have become a standard for formal specifying and reasoning about advanced transaction models. 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. None of the provided examples uses dependencies; instead, dependencies are implicitly modeled by flow control constructs. ASSET does not use the object paradigm for the proposed primitives; it rather uses procedural programming style.

PJama is a clone of the Java programming language that supports object persistency. In [20], the support for customizable transactions in PJama is introduced. Note that the current version of PJama still does not support extended transaction models in the way proposed in the paper. PJama introduces the `TransactionShell` class that provides basic transaction primitives, together with the `LockCapability`, `CollisionDetectGraph`, and `UpdateBookKeeper` classes that support locking, permitting, and storing, respectively. Custom transaction models could be provided using inheritance from the `TransactionShell` class, overloading its methods, or introducing new methods representing more advanced transaction primitives. 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.

8 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. Those events can represent transaction significant events and dependencies could be specified using the registration of methods reacting to some pre- or post- events. This model can be used for more advanced dependencies, such as dependencies on starting or finishing bean business methods, creating bean object, non-transactional dependencies, etc. 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 some open-source implementation of the EJB-based application server.

9 Conclusion

In this paper, we provide 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 some 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.

10 References

- [1] A. Biliris, S. Dar, N. Gehani, H.V. Jagadish, K. Ramamritham: ASSET: A System for Supporting Extended Transactions. Proceedings of ACM SIGMOD International Conference on Management of Data, May 1994
- [2] Panayiotis K. Chrysanthis: ACTA, A Framework for Modeling and Reasoning about Extended Transactions Models. Ph.D. Thesis, September 1991.
- [3] Jingshuang Yang, Gail E. Kaiser: JPernLite: Extensible Transaction Services for WWW. CUCS-009-98, Department of Computer Science, Columbia University, 1998
- [4] Vlada Matena, Mark Hapner: Enterprise Java Beans Specification 1.0. Sun Microsystems Inc., March 1998
- [5] Vlada Matena, Mark Hapner: Enterprise Java Beans Specification 1.1 Public Release. Sun Microsystems Inc., August 10, 1999
- [6] Linda G. DeMichiel, L. Ümit Yalçinalp: Enterprise Java Beans Specification 2.0 Draft. Sun Microsystems Inc., April 25, 2000
- [7] Seth White, Mark Hapner: JDBC 2.1 API. Sun Microsystems, October 5, 1999
- [8] Seth White, Mark Hapner: JDBC 2.0 Standard Extension API. Sun Microsystems, December 7, 1998
- [9] Mark Hapner, Rich Burrige, Rahul Sharma: Java Message Service API Specification 1.02, Sun Microsystems Inc., November 9, 1999
- [10] Susan Cheung, Vlada Matena: Java Transaction API 1.01 Specification. Sun Microsystems Inc., April 29, 1999
- [11] X/Open Distributed Transaction Processing: The XA Specification, 1991
- [12] Object Management Group: Object Transaction Service. December, 1997
- [13] Object Management Group: Concurrency Control Service. December, 1997
- [14] Susan Cheung: Java Transaction Service 1.0 Specification. Sun Microsystems Inc., December 1, 1999
- [15] Jim Gray, Andreas Reuter: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1993
- [16] Sushil Jajodia, Larry Kerchsberg: Advanced Transaction Models And Architectures. Kluwer, 1997
- [17] Ahmed K. Elmagarmid: Database Transaction Models For Advanced Applications. Morgan Kaufmann, 1992
- [18] Helmut Wachter, Andreas Reuter: The ConTract Model. In Ahmed K. Elmagarmid: Database Transaction Models for Advanced Applications, 1991
- [19] Garcia-Molina, H., Salem, K.: Sagas. In proceedings of the ACM SIGMOD Conference, 1987.
- [20] L. Daynes et al.: Customizable Concurrency Control for Persistent Java. In Advanced Transaction Models and Architectures, Editors: S. Jajodia , L. Kerschberg, August 1997

Appendix: BourgogneTransactions Interface

```
class TransactionSet extends AbstractSet {
    void add (javax.transaction.UserTransaction);
}

class BeanSet extends AbstractSet {
    void add (javax.ejb.EnterpriseBean);
}

interface BourgogneTransaction extends UserTransaction {

    // 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);

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

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

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

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

    // 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);

    // Give a set of transactions permission to invoke a method of a bean object
    void addPermission(TransactionSet, javax.ejb.EnterpriseBean,
        java.lang.method);

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

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

    // Give a set of transactions permission to invoke any method of any locked
```

```

// bean object
    void addPermission(TransactionSet);

// Remove a permission to invoke a method of a bean object
    void removePermission(BourgogneTransaction, javax.ejb.EnterpriseBean,
        java.lang.method);

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

// Remove a permission to invoke any method of a set of bean objects
    void removePermission(BourgogneTransaction, BeanSet);

// Remove a permission to invoke any method of any locked bean objects
    void removePermission(BourgogneTransaction);

// Remove a permission to invoke a method of a bean object
    void removePermission(TransactionSet, javax.ejb.EnterpriseBean,
        java.lang.method);

// Remove a permission to invoke any method of a bean object
    void removePermission(TransactionSet, javax.ejb.EnterpriseBean);

// Remove a permission to invoke any method of a set of bean objects
    void removePermission(TransactionSet, BeanSet);

// Remove a permission to invoke any method of a any acquired bean object
    void removePermission(TransactionSet);

// Remove any created permission
    void removePermission();
}

class BourgogneSet extends TransactionSet {
    void add (BourgogneTransaction);

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

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

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

// Remove a dependency on a set of transactions
    void removeDependency(TransactionSet, int);
}

```