Container-Interposed Transactions

Marek Procházka[1], František Plášil[1,2]

[1] Charles University, Faculty of Mathematics and Physics, Department of Software Engineering, Malostranské náměstí 25, 118 00 Prague 1, Czech Republic
{prochazka, plasil}@nenya.ms.mff.cuni.cz, http://nenya.ms.mff.cuni.cz

[2] Academy of Sciences of the Czech Republic, Institute of Computer Science, Pod vodárenskou věží 2, 180 00 Prague 8, Czech Republic
plasil@cs.cas.cz
http://www.cs.cas.cz

Abstract

The paper focuses on transaction context propagation in component-based software architectures, where every component is deployed into a container and client requests may be performed in the scope of a container-interposed transaction. In the existing commercial architectures such as EJB and COM+, the "single attribute approach" is used to specify the transaction context propagation. Being difficult to comprehend, this specification is also not powerful enough to express all the possible transaction propagation policies even for the flat transaction model; advanced transaction models cannot be employed at all. Moreover, the specification takes place as late as at the deployment time so that it is hard to reflect a particular transaction propagation policy in the component code. As a remedy, we propose to specify transaction propagation policy as a part of a component interface by means of a straightforward double attribute (NT&CT) approach. This way, advanced transaction models based on inter-transaction dependencies, giving permissions, and delegation can be also specified.[1]

Keywords: Transactions, component-based software architectures, transaction propagation policy, transaction attributes, container-interposed transactions

1. Introduction

In recent years, various component-based software architectures have been proposed in both academia ([2], [8], [13]) and software industry. In this field, the CORBA [17], Component Object Model (COM, [14], [15]), and Enterprise JavaBeans (EJB, [23]) technologies, all based on object-level, remotely accessible components belong to the key software industry players.

As opposed to the academic projects, in all these technologies, components can be involved in distributed transactions. The Object Transaction Service (OTS, [18]) was accepted as the standard for transaction processing in the distributed object environment. This standard adopts the basic X/Open XA distributed transaction model [26], defines interfaces of distributed transaction participants, and specifies some basic scenarios for executing transactions, namely the two-phase commit protocol. However, the standard does not address the transactions interposed/propagated by the environment wrapping a component. The key idea behind this concept, introduced in the Microsoft Transaction Server [12] (forming together with COM the COM+ technology) and later adopted by EJB, is that any client request potentially associated with a transaction is delegated to the desired component by its surrounding environment, which can modify the transaction context (e.g., it is able to suspend the client transaction and further create a new transaction and manage its demarcation).

In COM+ and EJB, *transaction propagation policy* is defined by the value of a single transaction attribute at the time of deploying components to the environment. Unfortunately, the attribute encodes several transactional factors and, therefore, is not easy to comprehend. As the first goal of the paper, we analyze this approach with the intention to propose a more flexible way to specify the transaction propagation policy (Section 2).

Moreover, in both COM+ and EJB, the specification of the transactional behavior of a component takes place as late as at the deployment time of the component. This contradicts to the natural requirement that the requested

[1] The research is partially supported by the Grant Agency of the Czech Republic (project number 201/99/0244), and the Grant Agency of the Academy of Sciences of the Czech Republic (project number A2030902). The results of the project will be also employed in the PEPiTA/ITEA project (the Eureka project number 2033).

transaction propagation policy should be known at the time of component implementation since this policy has to be reflected in the component's code (Section 2.2.4). Therefore, the second goal of the paper is to research the possibility to determine the transaction propagation policy as a part of the component interface specification.

Another weakness of the current component-based software architectures/products is that they do not support any other transaction model except for the flat transaction model; in particular this is true for both COM+ and EJB (the only exception is OTS supporting nested transaction). This is very limiting since today's applications are often long-living and require a higher level of transaction cooperation. In [22] we proposed the Bourgogne Transaction model that allows employing more advanced transaction models in EJB. In addition to the basic transaction primitives (`begin()`, `commit()`, `abort()`), Bourgogne Transactions provide advanced transaction primitives to establish inter-transaction dependencies, give permissions to share and to delegate enterprise bean instances among transactions. Based on the Bourgogne Transactions idea, the third goal of the paper is to propose a way to employ advanced transaction models in the container-interposed transaction settings and, at the same time, to specify the transaction propagation policy at the level of component interface (Section 3).

2. Container-interposed transactions in client-server settings

2.1 Basic concepts

A model scenario of a component-based application in client-server settings is shown in Figure 1 presenting a client and a set of components in a server.

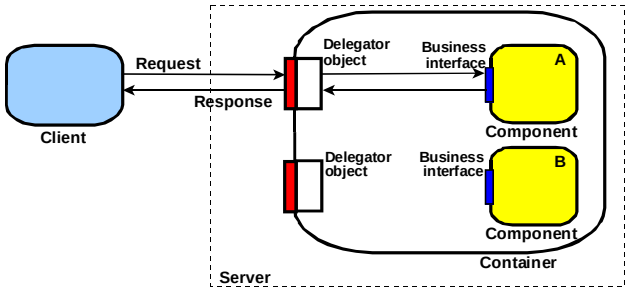

Figure 1. A Component-based application example

The components are deployed into a *container* that manages their lifecycle and provides other functionality, such as component lookup, security, persistence, and transactions. A number of components can be deployed into a single container; every component resides within exactly one container. A server may incorporate one or more containers. The *business interface* of a component is formed by the set of its (business) methods that can be invoked upon the *client call*. Each client call is interposed by the container in one of its *delegator object* (*delegator* for short) which delegates the call to a particular component so that it invokes the requested business method. A client call can be viewed as the sequence of a *client request*, a business method invocation, and a *component response*. Both the client request and component response are interposed by the delegator object.

To support transaction processing, a *transaction manager* is a part of the component-based application. In principle, a client call can be issued in the scope of a *client transaction* `tx`; technically, the corresponding client request implicitly transfers the *transaction context* which determines `tx` (Figure 2). In a typical scenario/implementation, the client first calls the transaction manager API and obtains a reference `tx` to a transaction object (representing a transaction). To start the transaction, the client calls `the begin()` method on `tx`; similarly `tx.commit()` resp. `tx.abort()` commits resp. aborts the transaction. The client can also set and get the state information of `tx` (e.g., set/get the timeout of `tx`, denote `tx` as rollback-only, etc.). In general, the scope of a transaction `tx` is determined by propagating the transaction context of `tx` via client calls among components.

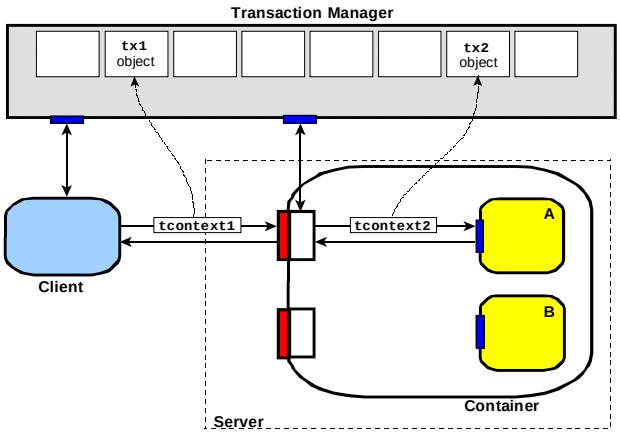

Figure 2. Transaction propagation

As a delegator object interposes every client request and component response, it in general modifies the transaction context `tcontext1` according to the *transaction propagation policy* (specified along with the component business interface) to `tcontext2` (Figure 2). For example the delegator can start a new transaction `tx2` (*container-demarcated transaction*) so that `tcontext1` ≠ `tcontext2`, or it can propagate the client transaction to the business method (`tcontext1 = tcontext2`). In general, the delegator can support resource sharing and delegation between the transactions determined by `tcontext1` and `tcontext2`; similarly, it can establish a dependency

between these transactions (such as the abort or commit dependency [[5]). In principle, a component A can issue a call to a business method of another component B. Then A plays the role of a client; it can start another client transaction txA so that the call will be done in the scope of txA. We say that the component A issued a *component-demarcated transaction* (Figure 3). In general, a transaction scenario, where the transaction context can be modified only by the delegator object of a container, we call *container-interposed transaction* scenario.

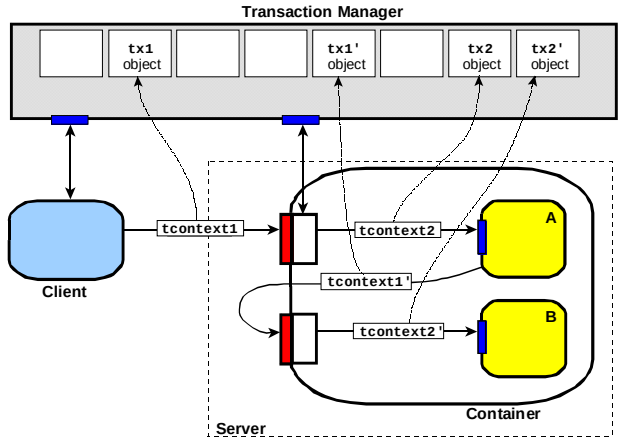

Figure 3. Container-interposed transaction

2.2 Simple flat transaction model

The simple flat transaction model supports two transaction context propagation policies: (1) All the business method invocations between a client transaction's begin and end are done within the scope of the client transaction – the client transaction context is *propagated* to the component, i.e. tcontext1 = tcontext2 in Figure 2 (transitively for all nested business methods calls). (2) The client transaction is suspended by the container (delegator). Naturally, the policy (2) means that the requested business method is not invoked in the scope of the client transaction and the delegator either creates a container-demarcated transaction, not known to the client, or the method is executed in the scope of no transaction. In the request, the delegator object replaces the transaction context tcontext1 by a new tcontext2 which is passed to the component, and resumes the client transaction tx1 when delegating the component response. In the flat transaction model, no resource sharing, resource delegation, nor dependences are considered.

2.2.1 Single attribute (Enterprise JavaBeans) approach: The transaction processing in the Sun's Enterprise JavaBeans (EJB, [23]) and Microsoft's Component Object Model (COM+, [12]) technologies is based on the simple flat transaction model in client-server settings as described in Section 2.2. In both technologies, the transaction propagation policy is determined by the value of a single *transaction attribute* associated with the invoked method. The transaction attribute is defined apart from the business interface specification and the component code, as late as in the deployment descriptor of the component. This is intended to give a diversity of transactional behavior options. An overview of the EJB transaction attribute values is provided in Table 1.

For example, if a method m is associated with the RequiresNew transaction attribute, m is always invoked in the scope of a newly created container-demarcated transaction. If a client invokes m with no transaction context, the container automatically starts a new container-demarcated transaction (referred to as *container-managed transaction* and also *declarative transaction* in EJB, and *automatic transaction* in COM+) before delegating a method call to m. (The EJB specification does not explicitly mention a delegator object; a request is simply interposed by "container".) If a client invokes m in the scope of a transaction tx, the container suspends tx before starting the new transaction txn in which it invokes m. The container resumes tx after m and txn have been completed. The container automatically enlists all the resource managers accessed by m within the newly created container-demarcated transaction txn. If m invokes a method mm on another bean, the container passes the transaction context of txn with the request. The container attempts to commit txn when m has completed.

Transaction attribute	Client's transaction	Transaction associated with business method	Transaction associated with resource managers
NotSupported	None	None	None
	T1	None	None
Required	None	T2	T2
	T1	T1	T1
Supports	None	None	None
	T1	T1	T1
RequiresNew	None	T2	T2
	T1	T2	T2
Mandatory	None	Error	N/A
	T1	T1	T1
Never	None	None	None
	T1	Error	N/A

Table 1. EJB Transaction attribute values

A very similar single attribute model is used in the Microsoft Transaction Server [12].

2.2.2 Analyzing the EJB approach: In EJB, the set of transaction attribute values is fixed and relatively small as the flat transaction model is very simple. If more advanced transaction models were to be supported, the set of transaction attribute values would have to be substantially enhanced. However, even for the current flat model, the semantics of a particular transaction attribute value is not easy to comprehend; the reason for it is that an attribute value encodes a combination of several transactional factors into one value - as the following analysis reveals:

(1) *Client transaction context handling*: It is necessary to determine how a client transaction is propagated to a component: whether an invoked business method is executed in the scope of the client transaction or not. The following alternatives are to be considered:
`Propagated`: The delegator propagates the transaction context to the component. The requested business method is executed in the scope of the client transaction.
`Suspended`: The transaction `tx` in which the client request was executed is suspended by the delegator.

(2) *A new (container-demarcated) transaction invocation*: It is to be specified whether a new transaction is created and started by the delegator. The alternatives to be considered are:
`NotCreated`: No container-demarcated transaction is created by the delegator.
`Created`: A new transaction `txn` is created by the delegator. The requested business method `m` will be executed in the context of `txn`. The transaction `txn` will be committed if `m` is successfully executed, or aborted if the execution of `m` fails for some reason.

(3) *Exception throwing*: It is necessary to determine whether an exception should be thrown by the delegator with respect to transaction handling. The alternatives to be considered are:
`IfClientTx`: An exception is thrown by the delegator if the client request is associated with a client transaction context.
`IfNotClientTx`: An exception is thrown by the delegator if the client request is not associated with a client transaction context.
`Never`: In any case, no exception is thrown at the time of the client request delegation by the delegator.

(4) *Relation between client and container-demarcated transactions*: This factor is to be considered if advanced transaction models (i.e., more advanced than the flat transaction model) are employed. A specified relation between the client transaction and the newly created container-demarcated transaction may have to be determined to reflect a particular transaction model. An option is to employ the primitives defined in the Bourgogne Transactions model proposed in [22]. This issue is discussed in Section 3 in a more detail.

By combining all the alternatives of the transactional factors 1, 2, and 3 above, we obtain all alternatives of transaction context propagation for the flat transaction model in client server settings. This is done in Table 2 which also indicates how the EJB transaction attribute values correspond to a particular combination.

The rows with combinations that do not make sense (5, 7, 10, 13, and 16) are highlighted. It is worth noticing that some of the combinations are not currently supported by EJB, even though they are meaningful in the flat transaction model (and, of course, in more advanced transaction models). For example, the rows 11 and 12 reflect a "multilevel" transaction model (not supported in EJB).

	Client tx	Client tx handling	Container-demarcated tx	Exception throwing	EJB 2.0 transaction attribute value	Two-attribute values approach
1	No	N/A	NotCreated	IfClientTx	*Never*	2, 1
2	No	N/A	NotCreated	IfNotClientTx	*Mandatory*	1, 3
3	No	N/A	NotCreated	Never	*NotSupported, Supports*	2, 3
4	No	N/A	Created	IfClientTx	not considered	3, 1
5	No	N/A	Created	IfNotClientTx	N/A	N/A
6	No	N/A	Created	Never	*Required, RequiresNew*	3, 3
7	Yes	Propagated	NotCreated	IfClientTx	N/A	N/A
8	Yes	Propagated	NotCreated	IfNotClientTx	not considered	1, 4
9	Yes	Propagated	NotCreated	Never	*Supports, Required*	2, 3
10	Yes	Propagated	Created	IfClient	N/A	N/A
11	Yes	Propagated	Created	IfNotClientTx	not considered	1, 5
12	Yes	Propagated	Created	Never	not considered	3, 5
13	Yes	Suspended	NotCreated	IfClientTx	N/A	N/A
14	Yes	Suspended	NotCreated	IfNotClientTx	not considered	1, 2
15	Yes	Suspended	NotCreated	Never	*NotSupported*	2, 2
16	Yes	Suspended	Created	IfClientTx	N/A	N/A
17	Yes	Suspended	Created	IfNotClientTx	not considered	1, 4
18	Yes	Suspended	Created	Never	*RequiresNew*	3, 4

Table 2.

2.2.3 NT&CT approach: We claim that the EJB transaction attribute values are not easy to comprehend, because they specify several factors that are orthogonal to each other (transaction context handling, new transaction invocation, exception throwing), including the fact whether a client transaction is or is not present, by encoding their combination into a single attribute value. To address this problem, we propose the *NT&CT* approach, based on specifying delegator behavior separately in two situations: (1) the client request is not associated with a transaction, and (2) the client request is associated with a transaction. To do so, we introduce two transaction attributes, `NoClientTransaction` (NT) and `ClientTransaction` (CT); the possible values of these attributes are listed in Table 3 and Table 4.

NT		
	Specified behavior	EJB 2.0 attribute settings
1	ThrowException	*Mandatory*
2	DoNothing	*Supports*, *NotSupported*
3	CreateNew	*Required*, *RequiresNew*

Table 3.

CT		
	Specified behavior	EJB 2.0 attribute settings
1	ThrowException	*Never*
2	Suspend	*NotSupported*
3	Propagate	*Supports, Required, Mandatory*
4	SuspendAndCreateNew	*RequiresNew*
5	Advanced	not considered

Table 4.

The last columns of both tables indicate which EJB transaction attribute values correspond to the particular NT, resp. CT, value. Note that, in the NT&CT approach, it is possible to specify a delegator behavior which is not supported by EJB. For example, by specifying `NT =`

`CreateNew` and `CT = ThrowException` we specify the combination 4 in Table 2.

2.2.4 Transactional behavior - when to be specified:

In EJB, every component is coded without any knowledge as to which transaction propagation policy will be used, because this policy is defined at the time of the component deployment. As a consequence, a client of the component is not able to determine the transaction propagation policy from the component interface. Similar approach is taken in COM+.

We believe that the transactional behavior of a component should not be defined as late as at its deployment time, but it should be determined at the time of the component's interface specification. This view can be justified as follows: The author of the component is responsible for its functionality and thus he/she has to reflect the desired transaction propagation policy in the code of each component method. Also, a client of the component should determine from its interface what transaction propagation policy will be applied when calling a particular method on the interface. Should revealing of the transactional policy be a security issue, we can imagine same kind of filtering/hiding of the NT and/or CT attributes to the client. Using the NT&CT approach, the transaction propagation policy could be included in the component interface specification in the way illustrated below:

```
interface BankAccount {
  int getBalance( ) {
    NT: DoNothing;
    CT: SuspendAndCreateNew;
  }
  int getHighestBalance( ) {
    NT: DoNothing;
    CT: Suspend;
  }
  void withdraw(int iAmount) {
    NT: CreateNew;
    CT: Propagate;
  }
  void deposit(int iAmount) {
    NT: CreateNew;
    CT: Propagate;
  }
}
```

Here, the transaction propagation policy is specified at the method granularity – for each method, a value of both the NT and CT attribute is specified. For example, if the `getBalance()`method is invoked in the scope of no client transaction, the delegator does not create any container-demarcated transaction and the `getBalance()` method is executed in the scope of no transaction. If `getBalance()` is invoked in a client transaction scope, the client transaction is suspended, a new container-demarcated transaction is created and `getBalance()` is executed in the newly created transaction scope. If no failure occurs during the `getBalance()`execution, the new transaction is committed and the client transaction is resumed. Similarly, if `withdraw()` or `deposit()` is invoked in the scope of no transaction, the delegator creates a new container-demarcated transaction and the method is invoked in the scope of the newly created transaction. If `withdraw()` or `deposit()` is invoked in the scope of a client transaction, the transaction is propagated, i.e., the respective method is executed in the scope of the client transaction.

3. Supporting advanced transaction models

Consider again the container-interposed transaction scenario from Figure 3. In EJB, `tx1` and `tx2` are always independent of each other (row 4 in Table 4.). To employ advanced transaction models in the same scenario, more specific relations between `tx1` and `tx2` should be specified. For example, `tx2` could be a child of `tx1` in the Nested Transaction model [9].

ACTA [5], a widely accepted formal framework for specifying transaction models, considers the abstraction views listed below to be the key building blocks of a transaction model (we provide their brief, mostly informal characteristics). Central to ACTA is also the notion of history of object events and significant event invocations by transactions. In principle, an object event is an invocation of an operation upon a data object (elsewhere also called a resource). A significant event is an invocation of a transaction processing primitive (such as transaction begin, commit and abort). Given a set TS of transactions, the history of TS is the partially ordered set of all the events associated with the transactions in TS. The partial ordering reflects the required temporal order of events.

(1) *Inter-transaction dependencies*. A dependency between two transactions `ti` and `tj` is a flow control obligation between the significant events of `ti` and `tj`; e.g., `tj` may be abort-dependent on `ti` which means that if `ti` aborts then `tj` also aborts.

(2) *Conflict relations between operations*. Two operations are said to be conflicting if their joint effect on the data they operate upon depends on the order in which they are executed. Typically, conflict relations are addressed by locking.

(3) *View of a transaction.* In principle, the view of a transaction at a particular time point determines the level of the mutual isolation of all transactions in terms of the visibility of the effects on data caused by them.

(4) *Conflict set of a transaction* is the set of such operations which effects have not been committed or aborted yet and that are conflicting with the operations involved in the transaction.

(5) *Delegation* of some operations from a transaction `tx1` to `tx2` means that all the object events having been exhibited by `tx1` will be considered as if they had been exhibited by `tx2`. As a consequence, the responsibility for committing, resp. aborting, these operations is transferred to `tx2`.

In the container–interposed transaction settings, such as EJB, one usually assumes the following: (a) The methods on a business interface are mutually excluded for any two threads, i.e. a component cannot be visited by more than one thread at a time. (b) A component once visited by a transaction `tx` is locked for other transactions until `tx` is finished. (c) ACTA object events are the invocations of business methods on component interfaces.

These assumptions very much predetermine the ACTA building blocks: ad (2) all operations (methods) on a business interface are conflicting if executed by separate threads; ad (3) the views of any two transactions are separated (because of (b)); ad (4) all the operations (methods) on the business interface of every component already visited by a transaction in progress are in the conflict set of the transaction.

Aiming at enhancing the EJB transactional functionality, we proposed in Bourgogne Transactions [22] several advanced transaction primitives to address inter-transaction dependencies, delegation, and resource sharing (to partially handle conflict relations between operations in the view of a transaction).

Below, we propose the Bourgogne Transaction advanced transaction primitives to be applied by the delegator to the transactions `tx1` and `tx2` in order to let these transactions cooperate in compliance with a desired transaction model assuming (a), (c) holds and (b) is eased in the sense that some methods can be exempt from the component lock. We will show that the required transactional action of the delegator (calls of the Bourgogne Transaction advanced transaction primitives implying new signification events) can be derived from the transactional attributes associated with the methods of a component interface.

Technically, we enhance the notation for specifying transaction attributes proposed in Section 2.2.3: The `CT` attribute set to `Advanced` (row 5 in Table 4) indicates that an advanced transaction model is employed. The required transaction model is specified by a 6-tuple of *CT subattributes*: `ClientDependency`, `ClientPermissions`, `ClientDelegate`, `CdtDependency`, `CdtPermissions`, and `CdtDelegate`. Here "`Client`" refers to the required action for a client transaction `tx1` and "`Cdt`" refers to the required action for a container-demarcated transaction `tx2`. Table 5 summarizes all these attributes and their possible values.

CT subattribute	Value
ClientDependency	None CommitDependency, StrongCommitDependency, AbortDependency, WeakAbortDependency, TerminationDependency, ExclusionDependency, ForceCommitOnAbortDependency, BeginDependency, SerialDependency, BeginOnCommitDependency, BeginOnAbortDependency, WeakBeginOnAbortDependency
CdtDependency	
ClientPermissions	None, All
CdtPermissions	None, A set of methods of the same interface
ClientDelegate	None, All
CdtDelegate	None, Always, BeforeCommit, BeforeAbort

Table 5.

Consider an example where a client transaction `tx1` invokes `Account.withdraw()` of the `Account` component. In the definition of the `Account` interface, the `withdraw()` method is associated with the `Advanced` CT attribute. The delegator object creates a new container-demarcated transaction `tx2`, which would behave like a child transaction in the Nested Transaction model (`tx1` is commit-dependent on `tx2`, `tx2` is weak-abort-dependent on `tx1`, `tx2` delegates to `tx1` all the operations `tx2` executed on the accessed components before commit, and `tx1` grants `tx2` the permission to access all the components that it has locked); moreover, `tx2` allows `tx1` to execute the `getBalance()` method of the invoked component. The advanced subattributes are defined as follows:

```
ClientDependency = CommitDependency;
CdtDependency = WeakAbortDependency;
ClientPermissions = All;
CdtPermissions = getBalance;
ClientDelegate = All;
CdtDelegate = BeforeCommit;
```

The corresponding delegator code fragment is shown below:

```
tx2 = getBourgogneTransaction( );
tx1.addDependency(tx2, CommitDependency);
tx2.addDependency(tx1, WeakAbortDependency);
tx1.addPermission(tx2, Account);
tx2.addPermission(tx1, Account, getBalance);
tx.begin();
try {
  Account.withdraw();
}
```

```
catch (Exception e) {
  tx2.abort();
  ...
}
tx2.delegate(tx1);
tx2.commit();
```

In Table 6 below, all the possible CT subattribute values are shown; for each of them, the corresponding Bourgogne Transaction primitive call is indicated.

CT subattributes	Bourgogne Transaction primitive
ClientDependency = dep	tx1.addDependency(tx2, dep)
CdtDependency = dep	tx2.addDependency(tx1, dep)
ClientPermissions = All	tx1.addPermission(tx2)
CdtPermissions = (m1, m2)	tx2.addPermission(tx1, I, m1) tx2.addPermission(tx1, I, m2)
ClientDelegate = All	tx1.delegate(tx2)
CdtDelegate = BeforeCommit CdtDelegate = AfterCommit CdtDelegate = Always	tx2.delegate(tx1)

Table 6.

Note that `I` stands for the interface in the definition of which the subattributes are specified, `tx1` is a client transaction, and `tx2` is a container-demarcated transaction created by the delegator. For the `CdtDelegate` subattribute, the `delegate()` method is invoked at the time before commit or abort for `BeforeCommit` and `BeforeAbort` values respectively, or it is invoked always before the container-demarcated transaction completion, no matter whether it is committed or aborted.

Below we describe the semantics of CT subattributes in more detail and illustrate a way they can be specified in component interfaces.

Dependencies. A dependency between two transactions `ti` and `tj` is a conditional flow control binding between the significant events of `ti` and `tj`; e.g., `tj` may be weak-abort-dependent on `ti` which means that if `ti` aborts than `tj` also aborts if it has not been committed yet. In the container–interposed transaction scenario, dependencies are specified via the `ClientDependency` and `CdtDependency` subattributes; their possible values in Table 5 reflect the basic twelve dependencies introduced in ACTA [5].

For example, in the definition of the `BankAccount` interface, the `CT` attribute of `withdraw()` method could be specified as `SuspendAndCreateNew`. Suppose a container-demarcated transaction `tx2` started by the delegator before the `withdraw()` execution should act as a nested transaction of a client transaction `tx1`, the weak-abort and commit dependencies of the `withdraw()` method have to be set as follows:

```
void withdraw(int iAmount) {
  NT: CreateNew;
  CT: SuspendAndCreateNew {
    ClientDependency = CommitDependency;
    CdtDependency = WeakAbortDependency;
  }
}
```

Resource sharing. In general, a transaction `tx1` can grant to another transaction `tx2` the permission for an access to a resource associated with `tx1` (locked by `tx1`). In the container-interposed transaction scenario (Figure 3), a client transaction `tx1` can give permissions to the container-demarcated transaction `tx2` to access the components (i.e. the resources) that `tx1` has locked; similarly, `tx2` can give to `tx1` the permission to access the called component via a subset of its business methods. Consider again the `withdraw()` method:

```
void withdraw(int iAmount) {
  NT: CreateNew;
  CT: SuspendAndCreateNew {
    ClientDependency = CommitDependency;
    CdtDependency = WeakAbortDependency;
    ClientPermissions = All;
    CdtPermissions =
              {getBalance, getHighestBalance};
  }
}
```

Here, by setting `ClientPermissions = All` any client transaction `tx1` gives the corresponding container-demarcated transaction `tx2` the permissions to access components that `tx1` has locked (`none` would indicate no such permission). Similarly, the `CdtPermissions` subattribute indicates that `tx1` can access the component via the `getBalance(), getHighestBalance()` methods even though the component is locked by `tx2`. Note that this can be employed only if the client transaction is multithreaded. As an aside, the semantics of this resource sharing can be easily explained via its implementation scenario – any method listed in a `CdtPermissions` subattribute is associated with a list of the transactions allowed to execute it.

Delegation. The delegation of operations as mentioned in the ACTA building block (5) can be advantageously expressed at the granularity of whole data objects/resources. Thus, a transaction `tx1` can delegate some of the resources associated with it to another transaction `tx2`, so that `tx2` becomes responsible for commit or abort of all the operations executed in `tx1` before the delegation took place. In the container-interposed transaction scenario (Figure 3), the `ClientDelegate` attribute specifies the delegation from a client transaction `tx1` to the corresponding container-demarcated transaction `tx2`. The values of `ClientDelegate` can be either `None` indicating that no component instance is delegated to the container-demarcated transaction by the client transaction, or `All` indicating that all component instances acquired by the

client transaction are delegated to the container-demarcated transaction, which is after the delegation responsible for committing or aborting modifications of the instances. The `CdtDelegate` attribute specifies delegation from the container-demarcated transaction to the client transaction. The values of `CdtDelegate` can be `None` indicating that nothing is delegated by the container-demarcated transaction, `BeforeCommit` or `BeforeAbort` indicating that all component instances acquired by the container-demarcated transaction are delegated to the client transaction before the transaction commit if the transaction is committed or before the transaction abort if the transaction is aborted; the value `Always` indicates delegation before the completion of the transaction, no matter whether it commits or aborts. For illustration, consider again the `withdraw()` method with the following delegation subattributes:

```
void withdraw(int iAmount) {
  NT: CreateNew;
  CT: SuspendAndCreateNew {
    ClientDependency = CommitDependency;
    CdtDependency = WeakAbortDependency;
    ClientPermissions = All;
    CdtPermissions = None;
    ClientDelegate = None;
    CdtDelegate = BeforeCommit;
  }
}
```

Here we specify `tx2` to be a nested transaction of the client transaction `tx1` since (a) there are mutual commit and weak-abort dependencies between these transactions, (b) `tx1` shares all of its resources with `tx2`, (c) before committing, `tx2` delegates all its resources to `tx1`. In a similar way, at the level of a component interface, behavior of the container-interposed transactions reflecting other advanced transaction models can be specified (e.g., Open Nested Transactions [10], Sagas [7], Split and Joint Transactions [21]).

4. Discussion

4.1 Evaluation

In this paper, we propose to specify the transaction propagation policy not as late as at the component deployment time, but at the time of defining the component interface since the author of the component has to design its code according to the propagation policy specified. Also a client of the component can determine from the interface how the component would participate in the client's transactions. Should revealing of the transactional policy be a security issue, we can imagine same kind of filtering/hiding of the transaction propagation policy to the client. As for the transaction propagation policy, we found the EJB/COM+ single attribute approach inappropriately complicated. We propose the NT&CT approach, where the transaction propagation policy is specified via distinguishing whether the component method is invoked (1) in the scope of no client transaction and (2) in the scope of a client transaction. In our view, this is more flexible/readable then the single attribute approach. In addition, the NT&CT attributes allow specifying additional policies, not covered by the EJB/COM+ transaction attributes.

To employ advanced transaction models, we apply the Bourgogne Transaction advanced transaction primitives in the delegator in order to let container-interposed transactions cooperate in compliance with a desired transaction model. We show that the required transactional action of the delegator (calls of the Bourgogne Transaction advanced transaction primitives) can be derived from the transactional attributes associated with methods of a component interface.

Specifying the transaction propagation policy in the component interface is quite different from specifying it in the component deployment descriptor. If the policy is defined as late as in deployment descriptor like in EJB, the author of the component code is not aware of the way in which transactions will be propagated to the component. This is one of the reasons why the EJB container always suspends the client transaction if a component-demarcated transaction is started in response for the client call. If the code of the component is written taking the interface and the transaction propagation policy into account (as in our approach), the delegator object does not always have to suspend the client transaction. In the proposed NT&CT approach, the value of the `ClientDependency` and `CdtDependency` subattributes can express one of the twelve predefined dependencies. Although they include those used most frequently, it might be desirable to specify additional dependencies (see Section 4.3).

4.2 Related Work

The impact of object-oriented technology on transaction processing is discussed in [6]. The author indicates new aspects of transaction processing applied to the environment of distributed objects and argues for extending object brokers (ORBs) by transaction processing monitors (TP monitors) scheduling functionality to form an *object TP monitor*. The paper, concentrating rather on the distributed object-based than the real component-based technology, does not address container-demarcated transactions, transaction propagation policy of a container, etc. As an extension to the OMG OTS, the authors of [1] address support of long-lived transactional computations.

In the framework of [19], researchers from the University

of Valenciennes propose to extend the transaction service of the JOnAS application server based on EJB. The extension supports the Nested and Open Nested Transaction models in EJB. Adopting the EJB concept of specifying the transaction propagation policy, the authors extend the set of transaction attributes by `RequiresNewSub`, `MandatoryNewSub` for nested transactions and `RequiresNewOpenSub`, `MandatoryNewOpenSub` for open nested transactions (thus following the original EJB single attribute approach – as opposed to our NT&CT approach). A support for other transaction models is not discussed.

In [16], the authors discuss the implementation of transactional business processes using components and indicate general requirements for so called *transactional business process servers*. A component-managed access to resources combined with declarative transactions is mentioned as a special case, but it is considered unnatural and potentially problematic to use, as "such facilities are not part of the programming or process modeling language". We believe NT&CT approach in the interface specification should be, at least, a partial remedy. As for distributed transactions, a consortium of key software technology vendors proposed the XAML language for distributed business-level transactions. At this point, the proposal does not go beyond a white paper explaining the basic motivation. A specification draft should follow shortly [25].

4.3 Future work

We are currently working on a prototype implementation of a transactional manager in order to apply the Bourgogne Transaction advanced primitives to transactions in the JOnAS EJB implementation (in the framework of the PEPiTA project [19]). Also, we intend to investigate the option of user-defined extension of the delegator subattributes. As a next step, we plan to research the option of employing the proposed transactional behavior specification in a more general component model with multiple provides and requires interfaces of potentially hierarchical components. In our opinion, a component exhibiting multiple provides interfaces, should be potentially shared among multiple transactions. As a proof of the concept, we intend to implement a transaction service for our SOFA/DCUP architecture [20], based on a hierarchical component model.

5. Conclusion

The paper is focused on container-interposed transactions. We analyzed the EJB-like single attribute approach for specifying the transaction propagation policy, found it hard to comprehend, and proposed the NT&CT approach based on specifying transactional behavior separately in two situations: (1) the client request is not associated with a transaction, and (2) the client request is associated with a transaction. We argue for specifying the transaction propagation policy not at the time of deploying a component, but as a part of the component interface.

To employ advanced transaction models in container-interposed transactions, we use the Bourgogne Transaction approach, which extends the classical transactional API for managing transaction demarcation by advanced primitives allowing establishing inter-transaction dependencies, component sharing, and delegation. We show how the delegator object derives calls of the Bourgogne Transaction advanced transaction primitives from the transactional attributes associated with the methods of a component interface.

Acknowledgement

The authors are grateful to the Joseph, Marie and Zdenka Hlavka Foundation for partially sponsoring our participation at the SNPD 2001 conference.

References

[1] Alcatel, IONA, IBM, Vertel, (University of Newcastle upon Tyne, Bank of America, INRIA, Bull), "Revised joint proposal for Additional Structuring for OTS" orbos/00-04-02, May 2000.

[2] R. J. Allen, "A Formal Approach to Software Architecture", Ph.D. Thesis, 1997.

[3] B. R. Badrinath, and K. Ramamrithan, "Semantics-Based Concurrency Control: Beyond Commutativity", ACM Trans. on Database Systems, vol. 17, no. 1, 1992, pp. 163-199.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman, "Concurrency Control and Recovery in Database Systems", Addison-Wesley, 1987.

[5] P. K. Chrysanthis, "ACTA, A Framework for Modeling and Reasoning about Extended Transactions Models", Ph.D. Thesis, Sep. 1991.

[6] E. E. Cobb, "The Impact of Object Technology on Commercial Transaction Processing", The VLDB Journal, vol. 6, no. 3, 1997, pp. 173-190.

[7] H. Garcia-Molina, and K. Salem: Sagas. In Proc. ACM SIGMOD Conference, San Francisco, CA, May 1987, pp. 249-257.

[8] D. Giannakopoulou, "Model Checking for Concurrent Software Architectures", Doctoral Dissertation, Imperial College, University of London, January 1999.

[9] J. Gray, and A. Reuter, "Transaction Processing, Concepts and Techniques", Morgan Kaufman, 1993.

[10] A. K. Elmagarmid, "Database Transaction Models For Advanced Applications", Morgan Kaufmann, 1992.

[11] S. Jajodia, and L. Kerchsberg, "Advanced Transaction Models and Architectures", Kluwer Publishing, 1997.

[12] S. Gray, R. Lievano, and R. Jennings, "Microsoft Transaction Server 2.0", Sams Publishing, 1997.

[13] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D.

Bryan, and W. Mann: Specification and Analysis of System Architecture Using Rapide, IEEE Transactions on Software Engineering, vol. 21, no. 4, Apr. 1995, pp. 336-355.

[14] Microsoft Corporation, “Component Object Model (COM) Specification 0.9”, Oct. 1995.

[15] Microsoft Corporation, “Distributed Component Object Model Protocol - DCOM/1.0”, Jan. 1998.

[16] T. Mikalsen, I. Rouvellou, S. Sutton Jr., S. Tai, M. Chessell, C. Griffen, and D. Vines, “Transactional Business Process Servers: Definition and Requirements”, in Proc. OOPSLA, Minneapolis, MN, Oct. 2000.

[17] Object Management Group, “The Common Object Request Broker: Architecture and Specification”, version 2.4.2, OMG document formal/01-02-33, Feb. 2001.

[18] Object Management Group, “The Transaction Service Specification”, version 1.1, OMG document formal/2000-06-28, May 2000.

[19] The PEPiTA/ITEA Project, “The PEPiTA Platform Architecture (Deliverable D1.1)”, version 1.0, Aug. 2000, http://www.objectweb.org/pepita/.

[20] F. Plasil, S. Visnovsky, and M. Besta, “Bounding Component Behavior via Protocols”, in Proc. IEEE TOOLS 30, Aug. 1999, pp. 387-398.

[21] C. Pu, G. Kaiser, and N. Hutchinson, “Split-Transactions for Open-Ended Activities”, in Proc. VLDB Conference, Los Angeles, CA, Sep. 1988, pp. 26-37.

[22] M. Prochazka, “Advanced Transactions in Enterprise JavaBeans”, in Proc. EDO 2000 Workshop, Davis, CA, Nov. 2000, pp. 208-223.

[23] Sun Microsystems Inc., “Enterprise JavaBeans Specification”, Version 2.0, Public Draft 2, Sep. 2000.

[24] Sun Microsystems Inc., “Java Transaction API Specification”, Version 1.01, Apr. 1999.

[25] XAML – Transaction Authority Markup Language, http://www.xaml.org/.

[26] X/Open Distributed Transaction Processing: The XA Specification, 1991.