A Flexible Framework for Adding Transactions to Components

Marek Prochazka
INRIA Rhône-Alpes
665, avenue de l'Europe, Montbonnot
38334 Saint Ismier Cedex, France
Marek.Prochazka@inrialpes.fr

Abstract: In this paper, we present a flexible framework for extending a component model with transactions. We first discuss key issues of combining components with transactions, and different approaches to achieve an appropriate level of transactional functionality in components. We distinguish between the *explicit* and *implicit* component participation approaches that differ by whether a component implements a part of transactional functionality or not. We discuss the influence of both approaches to concurrency control, recovery, and transaction context propagation. Then, we introduce our approach based on the interception pattern. We use several component *controllers* that manage transactional functionality on behalf of components. We identify several parts of transactional functionality that are implemented by different transactional controllers. For a component, to be *transactional*, the only requirement is to fulfill a *component contract* which is specific to various transactional controller implementations. We provide an overview of a prototype implementation of our approach in the Fractal component model. Thanks to the flexibility and reflective nature of Fractal, it is possible to achieve different levels of component transactional functionality by combining different transactional controllers, with only taking their component contracts into account. Our work proves that with an appropriate component framework that supports reflection and flexible component management with clearly defined notions of component composition, lifecycle, and binding, we can add transactions to components in an elegant and flexible way.

Keywords: Transactional components, concurrency control on components, transaction context propagation, interceptors

1   Introduction

During last years, a range of different component models has been proposed in both academia ([1], [5], [7], [14], [16]) and software industry ([8], [9], [12], [23]). As opposed to the majority of academic projects, in commercial technologies, components can be involved in distributed transactions. The ObjectWeb Consortium [13] is an open-source software consortium established in 1999. Its goal is the systematic development of open-source distributed middleware composed of flexible and adaptable components. In ObjectWeb, we believe that we need an appropriate basis for the construction of highly flexible, highly dynamic, heterogeneous distributed environments. Fractal [14], one of the ObjectWeb projects, provides a general software composition framework that supports component-based programming, including component definition, configuration, composition, and administration. The overall rationale for Fractal is the conviction that the engineering of technical software infrastructures for complex systems requires component models that are much more general that the existing industrial ones, together with open frameworks that allow for programmatic component management.

In the Java Open Transaction Manager project (JOTM, [15]), another ObjectWeb project, we aim at delivering software compatible with current standards such as the Java Transaction API (JTA, [24]) and CORBA Transaction Service (OTS, [10]), which would serve as the transaction manager for Enterprise JavaBeans (EJB, [23]), CORBA Component Model (CCM, [12]), Fractal, and other component middleware implementations. In this paper, we present a flexible framework for extending a component model with transactions. We first discuss key issues of combining components with transactions, and different approaches to achieve an appropriate level of transactional functionality in components. We distinguish between the *explicit* and *implicit* component participation approaches that differ by whether a component implements a part of transactional functionality or not. We discuss the influence of both approaches to concurrency control, recovery, and transaction context propagation. Then, we introduce our approach based on the interception pattern [22]. We use several component *controllers* that manage transactional functionality on behalf of components. We identify several parts of transactional functionality that are implemented by different transactional controllers. For a component, to be *transactional*, the only requirement is to fulfill a *component contract* which is specific to various transactional controller implementations. We have provided a prototype implementation of our approach in Fractal. Thanks to the flexibility and reflective nature of Fractal, it is possible to achieve different levels of component transactional functionality by combining different transactional controllers, with only taking their component contracts into account. We show that with an appropriate component framework that supports reflection as well as flexible component management with clearly defined notions of component composition, lifecycle, and binding, it is possible to add transactions to components in an elegant and flexible way.

The rest of the paper is organized as follows. In Section 2, we identify several issues of combining components and transactions and discuss various approaches to achieve transactional components. Section 3 provides an overview of the Fractal composition framework. In Section 4, we present our approach in general, together with implementation details of selected transactional controllers. An overview of related work is provided in Section 5 and we conclude in Section 6, where we also present our intentions for the future.

2 Combining components and transactions

The issue of adding transactions to components might seem to be an easy task; at least, there are several commercial standards and their implementations that deal with transactional components (e.g. EJB [23], CCM [12], .NET/COM+ [8], [6]). However, most of these standards employ simple ad-hoc solutions without clear identification of what the key issues of transactional components are. For example, this is the case with so called *transactional attributes* or concurrency control in EJB; it is not clear why transaction context propagation is defined by one of the seven predefined attribute values or why all components visited by a transaction are always covered by an exclusive lock. There are several research papers experiencing transactional components ([2], [3], [19]), but they are not related to a specific component framework and therefore their requirements on components are significantly vague. When speaking about transactional components, we have identified at least the following issues to solve: component participation in a transaction, concurrency control, recovery, and transaction context propagation. Let us discuss each of them separately in the following sections.

2.1 Component participation in a transaction

What does it mean that "a component takes part in a transaction" or that "a component is transactional"? Surprisingly, different transaction systems answer the question in different ways. An EJB component, for instance, is transactional thanks to 1) it synchronizes its persistent state at well-defined points in time and 2) it associates database connections with the current transaction. A CORBA object can be transactional if it implements the **Resource** or **Synchronization** interfaces in which case it can be registered to a transaction and take part in its two-phase commit. In [17], each component is supervised by a so called *spontaneous container*, which manages its persistence and participation in transaction in a similar way traditional databases do with ordinary data. In our opinion, all answers to the question what does it mean that a component takes part in a transaction could be divided into two main groups, depending on whether components take part in transactions implicitly or explicitly.

The *explicit transaction participation* is used in the current commercial component architectures. The scenario of involving a component **C** to a transaction **t** essentially consists of three steps:
1. **C** is *registered* to **t**. The transaction manager stores a pointer to **C**. More precisely, components usually implement various interfaces and, to be able to be registered to a transaction, a component has to implement a well-defined interface specific to a particular component architecture. For example, in the OTS, a transaction can enlist objects that implement one of the **Resource**, **SubtransactionAwareResource**, and **Synchronization** interfaces. In JTA, objects that implement the **XAResource** or **Synchronization** interfaces can be registered to a transaction.
2. The client invokes various operations on **C**. What makes the explicit transaction participation different from the classical database-like transaction paradigm is that the transaction manager is not aware of these operation executions: it has no scheduler.
3. At the time of **t**'s commit or abort, the transaction manager invokes specific methods of the registered **C**'s interfaces. The order of these invocations reflects a well-defined commit protocol, usually the two-phase commit protocol. For example, in OTS, the **prepare** and **commit**/**abort** methods of all registered **Resource** objects are invoked according to the two-phase commit protocol.

With the explicit transaction participation, scheduling of component operations (that correspond to data operations in database systems) is not driven by the transaction manager. Furthermore, the transaction manager is not aware of transaction effects and therefore is not responsible for their confirmation or cancellation by commit or abort, neither for their recovery in case of a system crash. All this functionality is instead implemented as a part of transactional components. Rather than ensuring ACIDity, standards used in the world of distributed components, like JTA and OTS, ensure only atomicity of sequences of operations invoked on objects registered to transactions[1].

On the other hand, the *implicit transaction participation* states that all the transactional functionality is driven by the transactional system itself. The scenario of involving a component **C** to a transaction **t** looks as follows:
1. Any time **C** is visited by a transaction **t**, the container keeps all necessary information to manage concurrency control, commit, rollback, and recovery.
2. When the client invokes any operation on **C**, the container is aware about it. It applies concurrency control protocols and controls **C**'s persistent state. The container behaves for components exactly like a database management system does for traditional data.
3. At the time of **t**'s commit or abort, the transaction manager commits or rollbacks all effects of **t** on **C** (and other components it manages). It can eventually take part in the two-phase commit of transactions that have been spread on multiple components supervised by different containers.

---

[1] However, JTA supports ACIDity through the use of XA resources; concurrency control and recovery is therefore provided at the level of an involved database. Both the JTA and OTS standards support ACID transactions but they do not provide all means to support all of the ACID properties and some of the properties should be ensured by other system components. For instance, concurrency control in CORBA can be managed by simple read/write locking in the Concurrency Control Service [11].

To summarize what is different between the implicit and explicit transaction participation, the former manages all the transactional functionality itself (therefore it seems to be implicit from the component's point of view), while the latter leaves the implementation of what happens at the time of commit, rollback or crash recovery on the code of transactional components (transactions are handled by components explicitly).

2.2 Concurrency control and recovery

Following the discussion of the previous paragraph, let us discuss what is different between the explicit and implicit transaction participation from the concurrency control point of view. As mentioned before, in the explicit transaction participation, the transaction manager does not support any implicit scheduling. For example, EJB use the JTA standard for transactions. However, JTA is not component-aware; JTA only provides an API for managing distributed transactions over XA resources in Java. As for concurrency control, JTA relies on the underlying database via JDBC connections. To have a locking policy at the component level, EJB add exclusive locking to any visited component. This approach makes it impossible to share any bean instance among transactions even if they are about to invoke read-only methods. In CORBA, any object visited by a transaction is not locked by default and applications can use the Concurrency Control Service (CCS, [11]) or other transaction-aware locking if needed. To summarize, there is no implicit concurrency control at the level of components in the explicit transaction participation. However, components can use an arbitrary transaction-aware concurrency control mechanism, such as CCS-like read/write locking in CORBA or mutual exclusion in EJB.

In the implicit transaction participation, concurrency control is supported by the transaction manager through the container that controls every component invocation. Similarly as in a database system, the container essentially implements the functionality of three entities: the transaction manager, scheduler, and data manager [4]. The transaction manager receives component operations (method invocations) and transaction operations (begin, commit, etc.) and forwards them to the scheduler. The scheduler ensures certain order of component and transaction operations. For each component operation sent by the transaction manager, the scheduler may 1) schedule it immediately by sending it to the data manager, 2) delay it by inserting it into some queue, or 3) reject it and cause the issuing transaction to abort. Various concurrency control policies, ranging from *aggressive* and *optimistic* schedulers that avoid delaying operations, to *conservative* schedulers tending to delay operations and avoid rejecting them, as well as different implementation techniques, based on e.g. locks, timestamps, or serialization graph testing, can be employed. A novel technique for ensuring correctness for general executions on transactional components has been introduced in [2].

As for recovery, the container supporting the implicit transaction participation manages recovery of every deployed component. On the opposite, in CORBA, a reference to a Recovery Coordinator is obtained when registering a resource to a transaction. A recoverable object has to use the Recovery Coordinator to drive the recovery process in certain situations. In EJB, recovery is supported only at the JDBC connection level. In principle, the container is able to continue two-phase commit on all the participating JDBC connections thanks to the XA protocol. There is no recovery protocol at the level of bean instances in EJB.

2.3 Transaction context propagation

To allow a component to participate in a client transaction, the *transaction context* has to be propagated from the client to the component. This implies the support for transaction context propagation in the communication protocol used (e.g., IIOP or RMI) and the container's ability to determine the transaction context from the client request. Along with the simple transaction propagation, more advanced manipulation of the transaction context can be provided. This includes applying various policies that specify for example whether an external transaction has to be present when invoking a particular method, whether a container creates a new (*container-managed*) transaction or the client transaction context is propagated to the component, etc.[20]. Essentially, there is no difference between the implicit and explicit transaction participation, since the propagation policy could be both set implicitly by the container or explicitly by the component author/deployer in both approaches. In EJB, CCM, and COM+, 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. Various frameworks that separate transaction demarcation from the container and allow defining new transaction propagation policies have been proposed ([19], [21]).

3 Fractal

In this section we give a brief overview of the Fractal Composition Framework [14]. In Fractal, a component is considered a run-time structure composed of two parts: a *controller* and a *content*. The content of a component is composed of (a finite number of) other components, which are under control of the controller of the enclosing component. The component model is recursive and allows components to be nested (i.e. to appear in the content of enclosing components) at an arbitrary level. Therefore, we distinguish a *primitive component* with no other component inside, and a *composite component* that contains other components. The notions of *subcomponent*, *parent component*, *child component*, and *top-level component* are used in their obvious way. Different components may share their

contents; i.e., a subcomponent may appear in the content of several components. A subcomponent that is shared among two or more components is controlled by their controllers.

A component interacts with its surrounding environment via its access points called *interfaces*. An interface is a set of operations (called *methods*) whose invocations reflect component interactions. Visibility of interfaces is managed by the component controller. A component may have multiple *server interfaces*, which define the functionality that the component offers to other components, and multiple *client interfaces*, which define the functionality the component requires from the surrounding environment. The controller of a component embodies the control behavior associated with a particular component. In particular, a component controller can intercept incoming and outgoing operation invocations and returns targeting or originating from the components in the component content. For example, the Lifecycle Controller manages the component lifecycle and the Binding Controller manages bindings to other components.

4   A flexible framework for adding transactions to components

4.1   Our approach

The overall architecture of a component extended with transactions is presented in Figure 1. There are several transaction controllers that manage specific transactional functionality. Most of this functionality is done in component interceptors that correspond to related transactional controllers. Each of the interceptors checks whether a transaction is associated with the thread asking for method invocation. Then the code specific to each interceptor is executed.
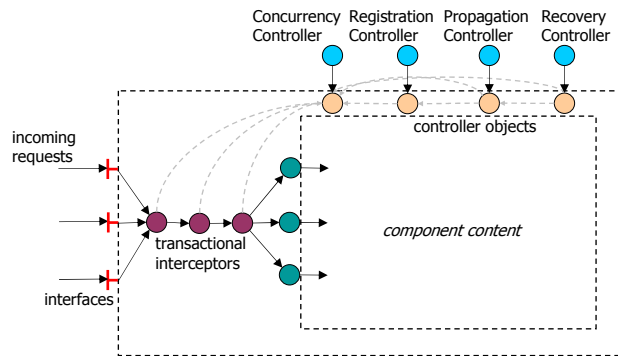

Figure 1: Transaction Controllers and Interceptors

We consider transactional controllers as follows:
–   **Propagation Controller**: Checks the transaction propagation policy and optionally modifies the transaction context.
–   **Registration Controller**: Allows participating components in transactions by registering transactional resources.
–   **Concurrency Controller**: Controls shared access to component instances.
–   **Recovery Controller**: Controls component recovery after a system crash.
The order of corresponding interceptors is important; e.g., the Propagation Controller has to interpose each operation invocation as the first controller in the interception chain, since it could modify the transaction context, suspend the original transaction and invoke the operation in the context of another transaction. We leave the decision whether to follow the explicit or implicit approach (Section 2.1) on particular controller implementations. For example, in our current prototype implementation, we use the explicit transaction registration along with implicit concurrency control and transaction context participation. Composite components allow us to use transactional controllers at any level of nesting; i.e., a subcomponent of a component can again be deployed with various transactional controllers and therefore can manage transactions in its own way.

The component's author does not care about the transactional controllers. When deploying a component, it is only necessary to specify which controllers and interceptors will be present in the target architecture. Moreover, the component's author eventually has to fulfill requirements of the controllers used. This is what we call the *component contract*. For instance, the OTS Registration Controller checks whether the component implements either the **Resource** or **Synchronization** interfaces. If it does, the interfaces are registered to the incoming transaction.

Till now, we have implemented only the Propagation, Registration, and Concurrency Controllers. We will discuss all of them in the following sections.

4.2   Concurrency Controller

We consider a component being a unit of concurrency control. For the Generic Concurrency Controller, we propose the interface shown in Figure 2.

```
1    GenericConcurrencyController {
2        boolean controlOperation(Transaction tx, String operation);
3    }
```

Figure 2: The interface of the Generic Concurrency Controller

If a component's method is invoked and the component is deployed with the Generic Concurrency Controller implementation, then the **controlOperation** method is invoked before the operation invocation. The **controlOperation** method can work as the scheduler described in Section 2.2; it can delay operation execution by temporarily suspending the current thread, reject the operation by returning **false** or e.g. rolling back the current transaction, or allow to schedule the operation by returning **true**.

Our current implementation of the Concurrency Controller uses the JOTM Lock Manager which supports transaction-aware locking with user-defined lock modes and user-defined conflict tables [15]. Let us have a component that implements a single **Account** interface with the **balance**, **deposit**, and **withdraw** methods. As **balance** is obviously not modifying the bank account balance, it is not conflicting with the other two methods. Even both **deposit** and **withdraw** modify the account balance, only **withdraw** is considered conflicting with other operations. The semantics here is that any dirty **balance** and concurrent **deposit** is correct, while multiple **withdraw** or **withdraw** combined with non-committed **deposit** can lead to negative balance of the bank account, which is considered undesirable. The corresponding conflict table is shown in Figure 3a, where "**–**" implies no conflict and "**+**" implies a conflict.

	balance	deposit	withdraw
balance	–	–	–
deposit	–	–	+
withdraw	+	+	+

```
1    <lock-controller>
2        <lock-mode name="balance">
3            <operation>Account.balance</operation>
4            <conflict held-mode="balance" value="false" />
5            <conflict held-mode="deposit" value="false" />
6            <conflict held-mode="withdraw" value="false" />
7        </lock-mode>
8        <lock-mode name="deposit">
9            <operation>Account.deposit</operation>
10           <conflict held-mode="balance" value="false" />
11           <conflict held-mode="deposit" value="false" />
12           <conflict held-mode="withdraw" value="true" />
13       </lock-mode>
14       <lock-mode name="withdraw">
15           <operation>Account.withdraw</operation>
16           <conflict held-mode="balance" value="true" />
17           <conflict held-mode="deposit" value="true" />
18           <conflict held-mode="withdraw" value="true" />
19       </lock-mode>
20   </lock-controller>
```

Figure 3a: Conflict table of the **Account** component example

Figure 3b: A Lock Controller configuration file with the balance, deposit, and withdraw lock modes

The only thing the author of a component has to do is to define a conflict table of methods of all implemented interfaces in the simple configuration file. This is the only component contract of the Concurrency Controller. The lock configuration in Figure 3b reflects locking with the conflict table in Figure 3a. Inside each lock mode definition (e.g. lines 2-7 for the **balance** lock mode), there is a list of methods associated with the lock mode (the **operation** element on line 3), together with the definition of conflicts (the **conflict** elements on lines 4-6). Obviously, an alternative configuration with the traditional read and write lock modes, where **balance** is associated with the read and both **deposit** and **withdraw** are associated with the write lock mode, could be easily defined. The Concurrency Controller works as follows:
  – It is initialized during the application instantiation. During the initialization, the XML configuration file is parsed to detect which interfaces and methods are subjects to locking. There is always a single lock associated with each component. The lock modes are values of **lock-mode** elements and the conflicts are attribute values of **conflict** elements in the configuration file. Also, the Concurrency Controller is registered as a transaction participant.
  – For each request for method invocation, the Concurrency Interceptor detects whether a transaction is associated with the request.
  – If a transaction is associated with the request, the Concurrency Interceptor finds whether the invoked method and interface are subjects for locking. This is true only if the method name was defined in the **operation**

5

element of one of the lock modes in the configuration file.
- If yes, the JOTM **TransactionLock** is acquired in the mode corresponding to the invoked method (the name of the lock mode in whose definition the method was listed in the **operation** element).
- The lock is released when the transaction is about to commit or abort. This is possible due to the fact that the Concurrency Controller has been registered as a transaction participant during the initialization phase.

Thanks to the Concurrency Controller based on the JOTM locking with an arbitrary conflict table the author of a component can exploit methods' semantics and therefore the component sharing potential is increased comparing to the traditional read/write approach. Moreover, the controller configuration file is easy to define (single conflict table per a component type). Our current Concurrency Controller implementation uses the implicit approach (i.e., the component lock is controlled by the controller), but the lock configuration is up to the component's author.

4.3   Registration Controller

The Registration Controller manages the registration of components to transactions. We have decided to follow the explicit transaction participation approach, especially due to the fact that we do not have any appropriate container being able to manage implicit transaction participation, component persistency, and recovery. So far we have implemented three registration controllers that are able to register a component to one of the OTS, JTA, and JOTM transactions. For every issued method invocation, the interceptor of the corresponding registration controller detects whether a transaction is associated with the request and whether it is the transaction's first visit of the component. If an OTS, JTA, or JOTM transaction is associated with the request, the corresponding Registration Interceptor finds whether the component implements the **org.omg.CosTransactions.Resource** and **org.omg.CosTransactions.Synchronization** OTS interfaces, the **javax.transaction.xa.XAResource** and **javax.transaction.Synchronization** JTA interfaces, or the **org.objectweb.jotm.core.events.EventListener** JOTM interface. If yes, the respective interface is registered to the issuing transaction.

The required component contract of the OTS, JTA, and JOTM Registration Controllers is to implement the corresponding interfaces. To allow components to register other types of transactional resources, we have also implemented the Generic Registration Controller. To be able to participate in a transaction, a component has only to implement the **ClientInterceptor** interface as presented in Figure 4.

```
1   public interface ClientInterceptor {
2       public void registerResources(org.objectweb.jotm.core.transaction.BasicTransaction tx);
3       public void registerResources(javax.transaction.Transaction tx);
4       public void registerResources(org.omg.CosTransactions.Control tx);
5   }
```

Figure 4: The **ClientInterceptor** interface

For every request for method invocation, the Generic Registration Interceptor detects whether a transaction is associated with the request and whether it is the transaction's first visit of the component. If yes, it finds whether the component implements the **ClientInterceptor** interface or not. If so, the corresponding **ClientInterceptor.registerResources** method is invoked depending on the type of the transaction associated with the invoking thread. With the Generic Registration Controller, the registration of transaction participants is not done by the interceptor, but by the component itself. In other words, it is up to the component author, which resources will be registered to a transaction that visits the component.

4.4   Propagation Controller

As described in Section 2.3, the Propagation Controller is responsible for the propagation of transaction context to the component. The behavior of the controller is driven according to various transaction propagation policies. In principle, the Transaction Propagation Interceptor is able to modify the transaction context in which scope the requested component method is invoked. In the current version of our prototype, the Propagation Controller has only a simple policy for transaction propagation: If there is no transaction associated with the client request, the requested component method is not executed in the context of a transaction. If a transaction is associated with the client request, the transaction is propagated to the component. In the future, we expect to have the transaction propagation policy specified in a configuration file similarly to the Concurrency Controller configuration. We plan to reuse the Transaction Demarcation Framework [21], which has been proposed as a part of JOTM/ObjectWeb.

4.5   Fractal use case

In Figure 5, you can see a trace of JOTM when used in transactional controllers with a Fractal component. Lines 1-3 are not related to transaction controllers: an OTS transaction is started. Line 4 starts the trace related to the Propagation

Controller. It detects that the **balance** method invocation has been requested and propagates the transaction to the component. On lines 5-9, an OTS transaction has been detected and, through reflection, it has been determined that the component implements the OTS **Synchonization** interface. The interface is registered to the transaction on line 9. Similarly, lines 10-12 reflect the registration of the **Resource** interface. Lines 13-15 reflect the Concurrency Controller task; it has found the **balance** lock mode associated with the **balance** method (line 13). It has found that no lock has been acquired yet on the component (line 14), and finally locks the **TransactionLock** in the **balance** mode (line 15).

```
1       0 [tm ]   DEBUG - OtsTx: created
2       2 [tm ]   DEBUG - OtsTx: begin invoked
3       5 [rec]   DEBUG - LogWriter: Associate tx to log file
4       7 [rec]   DEBUG - Account.balance method invoked
5      10 [ots]   DEBUG - OTS transaction interceptor preMethod launched
6      10 [ots]   DEBUG - OTS transaction detected
7      12 [ots]   DEBUG - Synchronization interface found
8      13 [ots]   DEBUG - SynchronizationParticipant: Created
9      14 [ots]   DEBUG - Register Synchronization
10     14 [ots]   DEBUG - Resource interface found
11     19 [ots]   DEBUG - ResourceParticipant: Created
12     19 [ots]   DEBUG - Register Resource
13     41 [cc ]   DEBUG - Lock: Asked lock balance by OtsTx
14     41 [cc ]   DEBUG - Lock: No lock currently acquired
15     41 [cc ]   DEBUG - Lock: balance acquired by OtsTx
```

Figure 5: An example of the JOTM when intercepting the **balance** method of an **Account** component

5   Related work

Comparing to the current commercial component architectures, such as EJB, COM, or CCM, an important feature of the Fractal transaction controllers is that they can be flexibly combined. Thanks to the flexibility and reflective nature of Fractal, it is possible to achieve different levels of component transactional functionality. The only requirement is that component contracts of combined transactional controllers can not conflict and the interception order has to be taken into account. It is for example possible to use concurrency control at the level of components if JTA transactions are used or to add EJB-like transaction propagation policies to OTS. The only requirement with respect to the component implementation is to fulfill the component contracts of the employed controllers. In the Registration Controller case, it means to implement one of the interfaces that can be registered to a transaction. The Concurrency Controller requires to define the lock modes and the Propagation Controller requires to define propagation policies in a configuration file, but both controllers can also use default values, such as exclusive locking and simple transaction propagation.

Our current implementation does not support dynamic interceptors being able to be added or removed to the interception chain. The authors of [17] use dynamic aspects to extend a component with a transactional functionality at runtime, which is useful especially in mobile environments. Our aim is also to extend our prototype with runtime adaptability. We have found our approach very close to the aspect-oriented one. However, we believe that thanks to the clearly defined notions of component composition, component lifecycle reflected by the Lifecycle Controller, component binding reflected by the Binding Controller, and thanks to the Fractal reflective architecture, we could address some of the most problematic issues related to dynamic aspects, e.g. a composition of multiple aspects, in a more elegant and flexible way.

6   Conclusion

In this paper, we have presented a flexible framework for extending a component model with transactions. We have discussed several key issues of combining components with transactions, as well as different approaches to address these issues. Then, we have introduced our approach based on the interception pattern. We use several component controllers that manage different parts of transactional functionality on behalf of components. For a component, to be transactional, the only requirement is to fulfill a component contract which is specific to various transactional controller implementations. We have provided an overview of a prototype implementation of our approach in Fractal. Thanks to the flexibility and reflective nature of Fractal, it is possible to achieve different levels of component transactional functionality by combining different transactional controllers, with only taking their component contracts into account. Our work has shown that having an appropriate component framework that supports reflection and flexible component management with clearly defined notions of component composition, lifecycle, and binding, we can extend transactions with components in an elegant and flexible way.

In the future, we would like to implement a Recovery Controller and various Propagation Controllers that will use a controller-managed transaction to form together with a client transaction a parent-child pair in the nested or a relation in the split/join transaction models. Another interesting task is to make the Fractal controllers dynamic to allow to add interceptors at runtime. Relations of transactional controllers to other Fractal means, such as the Binding Controller, Lifecycle Controller, or subcomponents shared among multiple components, have to be studied in more detail.

References

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

[2] Alonso, G., Fessler, A., Pardon, G., Schek, H.-J., "Correctness in General Configurations of Transactional Components", in Proceedings of the ACM Symposium on Principles of Database Systems (PODS '99), Philadelphia, Pennsylvania, USA, 1999

[3] Andersen, A., Blair, G., Goebel, V., Karlsen, R., Stabell-Kulø, T., Yu, W., "Arctic Beans: Configurable and Reconfigurable Enterprise Component Architectures", IEEE Distributed Systems Online, Vol. 2, No. 7, http://dsonline.computer.org/, 2001

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

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

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

[7] Luckham, D., C., Kenney, J., J., Augustin, L., M., Vera, J., Bryan, D., Mann, W., "Specification and Analysis of System Architecture Using Rapide", IEEE Transactions on Software Engineering, vol. 21, no. 4, Apr. 1995, pp. 336-355

[8] Microsoft Corporation, "Component Object Model (COM) Specification 0.9", October 1995.

[9] Microsoft Corporation, "Distributed Component Object Model Protocol - DCOM/1.0", January 1998.

[10] Object Management Group, "Transaction Service", Version 1.2, formal/01-05-02, May 2001

[11] Object Management Group, "Concurrency Control Service", Version 1.0, formal/00-06-14 April 2000

[12] Object Management Group, "CORBA Component Model", ptc/2001-11-03, November 2001

[13] ObjectWeb, http://www.objectweb.org/

[14] ObjectWeb, "The Fractal Composition Framework Specification", version 1.0, http://www.objectweb.org/fractal/

[15] ObjectWeb, "The Java Open Transaction Manager", http://www.objectweb.org/jotm/

[16] Plasil, F., Visnovsky, S., "Behavior Protocols for Software Components", IEEE Transactions on Software Engineering, vol. 28, no. 11, Nov 2002

[17] Popovici, A., Alonso, G., Gross, T., "Spontaneous Container Services", European Conference on Object-Oriented Programming, to appear, July 2003

[18] Prochazka, M., "Advanced Transactions in Enterprise JavaBeans", in Proceedings of the Engineering Distributed Objects (EDO) Workshop, Davis (CA), November 2000

[19] Prochazka, M., Plasil, F., "Container-Interposed Transactions", in Proceedings of the Component-Based Software Engineering (CBSE) Special Session of the SNPD '01 Conference, Nagoya, Japan, August 2001

[20] Prochazka, M., "Advanced Transactions in Component-Based Software Architectures", Ph.D. thesis, Charles University, University of Evry, February 2002

[21] Rouvoy, R., Merle, P., "Abstraction of Transaction Demarcation in Component-Oriented Platforms", ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, to appear, June 2003

[22] Schmidt, D. C., Stal, M., Rohnert, H. Buschmann, F., "Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects", John Viley & Sons, pp. 666, 2000

[23] Sun Microsystems, "Enterprise JavaBeans Specification", Version 2.0, Final Release, August 2001

[24] Sun Microsystems, "Java Transaction API (JTA)", Version 1.01, April 1999

[25] X/Open Distributed Transaction Processing: Reference Model, Version 3, February 1996