Jironde: A Flexible Framework
for Making Components Transactional

Marek Prochazka

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

Abstract. It is generally agreed that one of the key services of component-based systems are transactions. However, an agreement on how components should be involved in transactions is still missing. In this paper, we discuss some of the 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 use of several component controllers that manage 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 different 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 make components transactional in an elegant and flexible way.

1 Introduction

During last years, a range of different component models has been proposed in both academia [1, 5, 8, 15, 17] and software industry [9, 10, 13, 24]. It is generally believed that transactions belong to key services of component-based systems. However, there is no agreement on how the transaction service should look like and how components should be involved in transactions. The goal of the paper is twofold:

– We discuss some of the key issues of combining components with transactions, as well as different approaches to achieve transactional 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 Jironde, a flexible framework for making components transactional. Jironde uses several *component controllers* that manage transactional functionality on behalf of a component. 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 different transactional controller implementations. We present an overview of the Jironde prototype implemented using the Fractal Composition Framework [15].

The main contribution of Jironde is its flexibility. Combining different transactional controllers makes it possible to achieve different levels of component's transactional functionality, as well as to employ different transactional standards. The only obligation for components is to take component contracts of used transactional controllers into account. We show that with an appropriate component framework that supports reflection as well as flexible component management with well 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 and different approaches for combining components and transactions. In Section 3, we introduce our Jironde framework for making components transactional. Section 4 gives details of the Jironde prototype implementation in the Fractal environment. An evaluation and overview of related work is provided in Section 5 and we conclude with Section 6, where we also present our plans 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 standards and architectures that deal with transactional components, e.g. Enterprise JavaBeans (EJB, [24]), CORBA Components (CCM, [13]), and the Component Object Model (COM, [9, 10]) forming together with the Microsoft Transaction Server [6] the COM+ technology. However, most of these standards employ simple ad-hoc solutions without addressing key issues of transactional components. Several research papers have investigated transactional components [2, 3, 21], but their visions of transactional components significantly differ. 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 component systems answer these questions differently. An EJB component, for instance, is transactional thanks to the EJB container that 1) synchronizes the component's persistent state at well defined points in time and 2) associates database connections opened by the component with the current transaction. A CORBA object takes part in transactions if it implements one of the `Resource`, `Synchronization`, or `SubtransactionAwareResource` inter-

faces in which case it can be registered to a transaction and participate in its two-phase commit. In [18], 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 approaches could be divided into two main groups, depending on whether components take part in transactions implicitly or explicitly.

The scenario of involving a component *C* to a transaction *t* in the *explicit transaction participation* essentially consists of three steps:

1. *C* is registered to *t*. The transaction manager stores a pointer to *C*. More precisely, 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 Java Transaction API (JTA, [25]), 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 selected methods of the registered *C*'s interfaces. The order of these invocations reflects a well defined commit protocol. For example, in the CORBA Transaction Service (OTS, [11]), 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, such as OTS and JTA, ensure only atomicity of sequences of operations invoked on objects registered to transactions[1].

In the *implicit transaction participation*, all transactional functionality is implemented by the container and its transaction manager themselves. 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 a client invokes any operation on *C*, the container is aware of it. It applies concurrency control protocols and manages *C*'s persistent state. The container behaves for components exactly like a database management system does for traditional data.

---

[1] However, JTA supports ACIDity through the use of XA resources [26]. Concurrency control and recovery is therefore provided at the level of the databases involved. 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 through a simple read/write locking as specified in the Concurrency Service [12].

3. At the time of *t*'s commit or abort, the transaction manager commits or rollbacks all effects of *t* on *C* (as well as other components it manages). It can eventually take part in two-phase commit of transactions spread on multiple components supervised by different containers.

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 in 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 and only provides an API for managing distributed transactions over XA resources [26] in Java. As for concurrency control, JTA relies on the underlying databases 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 CORBA Concurrency Service (CCS, [12]) or other transaction-aware locking if needed.

To summarize, there is no implicit concurrency control at the level of components if the explicit transaction participation is used. 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.

With 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 a 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 to avoid rejecting them, as well as different implementation techniques, based on e.g. locks, timestamps, or serialization graph testing, can be employed.

As for recovery, the container supporting the implicit transaction participation manages recovery of every deployed component. On the opposite, with the explicit transaction participation, components should manage their recovery themselves, since

the container does not have enough information to do that. For example, 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 the 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. [21].

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 [20, 23].

3   Jironde

Jironde is a framework for making components transactional. The architecture of a component enabled to participate in transactions is shown in Fig. 1. The key ideas behind Jironde are as follows:

- To be transactional, a component is extended by several *transactional controllers* that manage transactional functionality on its behalf.
- The functionality implemented by transactional controllers is not fixed or determined by a transactional standard used. Instead, each transactional controller may implement a part of the transactional functionality (e.g., concurrency control) in its own way.
- The set of transactional controllers used by the component is specified during the component deployment. In other words, the way transactions are managed by the component is determined by its deployment configuration.
- The component must fulfill the *component contracts* of all the transactional controllers used in order to manage transactions correctly.
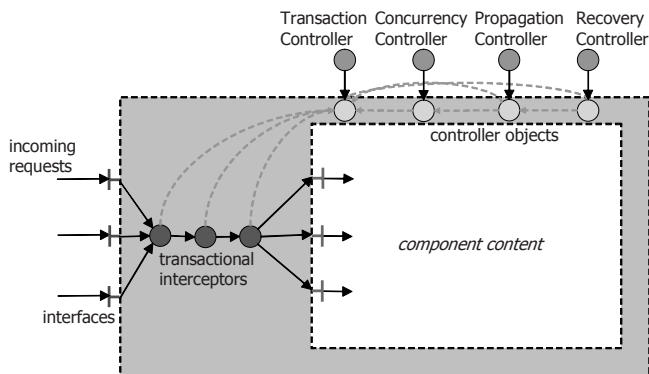
Fig. 1. The architecture of a component enhanced to support transactions. The controller part of the component contains several transactional controllers that manage transaction participation on behalf of the component

We have borrowed our terminology from Fractal, a generic composition framework based on components, which we have also used for a prototype implementation of Jironde (described in Section 4). However, Jironde is not Fractal-specific and can work with any component model that supports reflection, flexible component management through a configurable set of controllers[2], as well as means supporting invocation interception.

With having a configurable set of transactional controllers, 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 participation along with implicit concurrency control and transaction context propagation. 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 transactional controllers. When deploying a component, it is only necessary to specify which controllers will be present in the target architecture. Obviously, this approach gives component developers/deployers a big portion of flexibility, but the specification of a deployment configuration as well as combining different transactional controllers have to be handled with care.

To participate in transactions correctly, a component has to fulfill requirements of the controllers used – the component contracts. For example, the OTS Transaction Controller in our prototype checks whether the component implements either the `Resource` or `Synchronization` interfaces. If it does, the interfaces are registered to the current OTS transaction. 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.

---

[2] By a *controller* here we mean any runtime entity that provides certain management services for a component. It may have a form of a wrapper, interceptor, adapter, proxy, and others.

4 Implementing Jironde in Fractal

Jironde has been made as a part of the Java Open Transaction Manager (JOTM, [16]), one of the ObjectWeb [14] projects. Fractal [15], another ObjectWeb project, provides a general software composition framework that supports component-based programming, including component definition, configuration, composition, and management. In the next section we provide a brief overview of Fractal, and then we present details of our Jironde implementation in Fractal.

4.1 The Fractal Composition Framework

In Fractal, a component is considered a run-time structure composed of two parts: a *controller part* and a *content part*. The content part (or 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 the obvious way.

A component interacts with its surrounding environment via its access points called *interfaces*. An interface is a set of *methods* whose invocations reflect component interactions. Visibility of interfaces is managed by the component controller part composed of several *controller objects* (or *controllers* for short). 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 part of a component embodies the control behavior associated with this 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.

4.2 Concurrency Controller

In the Jironde prototype, a component is a unit of concurrency control. The Concurrency Controller works as a database scheduler. It can delay the method execution by temporarily suspending the invoking thread, or reject the method invocation by rolling back the current transaction, or schedule the method invocation.

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, which may define both non-symmetric and non-transitive conflict relations [16]. Let us have an example 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 bal-

ance, only `withdraw` is considered conflicting with other operations. The semantics here is that any dirty balance retrieved due to concurrent deposits is considered 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 Table 1, where "−" implies no conflict and "+" implies a conflict.

Table 1. The conflict table of the `Account` component example

	balance	deposit	withdraw
balance	−	−	−
deposit	−	−	+
withdraw	+	+	+

The only thing the author of a component has to do is to define a conflict table of methods of all implemented interfaces in a simple configuration file as follows:

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

The configuration file is the only component contract of the Concurrency Controller. The lock configuration above reflects the conflict table in Table 1. Inside each lock mode definition (e.g., lines 3-5 for the `balance` lock mode) is a list of methods associated with the lock mode (the `operation` element on line 4), together with the definition of conflicts (the `conflict` elements on lines 8 and 12-14). The default conflict value is `false` (line 2) and therefore all methods not explicitly listed in the `conflict` element are considered non-conflicting. An alternative configuration with the traditional read/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 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 is a subject for locking. This is true only if the method name was defined in the `operation` element of one of the lock modes in the configuration file.
- If the method is subject for locking, the JOTM Transaction Lock 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 at the time of transaction commit or abort. This is possible thanks 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 method semantics and therefore the component sharing potential is increased comparing to the traditional read/write approach. The controller configuration file is easy to define (a 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 developer/deployer.

4.3   Transaction Controller

The Transaction 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. We have implemented three Transaction Controllers that are able to register a component to one of the OTS, JTA, and JOTM transactions, respectively. Each Transaction Controller implementation allows to get a transaction object to demarcate transactions, to get the current transaction, and to configure transactions in the component according to the underlying standard.

For every issued method invocation, the interceptor of the corresponding Transaction 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 Transaction Controller finds whether the component implements the `Resource` and `Synchronization` OTS interfaces, the `XAResource` and `Synchronization` JTA interfaces, or the `EventListener` JOTM interface. If yes, the respective interface is registered to the issuing transaction.

The required component contract of the OTS, JTA, and JOTM Transaction Controllers is to implement the corresponding interfaces. To allow components also to register other types of transactional resources, we have also implemented the Generic Transaction Controller. Its component contract states that the component takes the responsibility for the resource registration and implements the `Registration` interface. When intercepting a method invocation, the Generic Transaction Controller finds out if the component implements `Registration` – in this case the `Registration.registerResources` method is invoked. It is up to the component which resources are in this method 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. When intercepting a method invocation, the Propagation Controller 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 similar to the Concurrency Controller one. We plan to employ the Open Transaction Demarcation Framework [23], which has been proposed as a part of JOTM/ObjectWeb.

5   Evaluation

Comparing to the current commercial component architectures, such as EJB, COM+, or CCM, an important feature of Jironde is its flexibility. Combining different transactional controllers makes it possible to achieve different levels of component's transactional functionality as well as to employ different transactional standards. 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 our Fractal prototype implementation of Jironde, for the Transaction Controller 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 [18] 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 such a runtime adaptability. We have found our approach very close to the aspect-oriented

one. We agree with conclusions in [7] that transactions are hard to aspectize, especially when aspectizing constructs of a programming language. However, it seems to us that aspectizing a well defined architectural and programming framework makes things different. We believe that thanks to the well defined notions of component composition (reflected in Fractal by the Content Controller), component lifecycle (Lifecycle Controller), component binding (Binding Controller), and thanks to the use of reflection, we can address some of the most problematic issues related to aspects (e.g., a composition of multiple aspects).

6   Conclusions

In this paper, we have discussed some of the key issues of combining components with transactions, as well as different approaches to make components transactional. We have introduced Jironde, a flexible framework for making components transactional. The main contribution of Jironde is its flexibility. Thanks to managing transactions in components by a configurable set of transactional controllers, it is possible to achieve different levels of component's transactional functionality, as well as to employ different transactional standards. In the future, we would like to study in more detail the collaboration of transactional controllers with other ones, such as the Binding Controller and Lifecycle Controller. We plan to identify more precisely Jironde requirements on component architectures, as well as limitations of combing transactional controllers with incompatible component contracts. As for the Jironde prototype in Fractal, we plan 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 transaction model, or a relation in the split/join transaction model. Another interesting task is to employ dynamic interceptors, which have been recently added to Fractal, to enable transactional controllers to add or remove their interceptors at runtime.

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, 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 (1999)
6. Gray, S., Lievano, R., Jennings, R., "Microsoft Transaction Server 2.0", Sams Publishing (1997)

7. Kienzle, J., Guerraoui, R., "AOP: Does it Make Sense? The Case of Concurrency and Failures", in Proceedings of the 16th European Conference on Object-Oriented Programming, Malaga, Spain (2002)
8. 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 (1995) 336-355
9. Microsoft Corporation, "Component Object Model (COM) Specification 0.9" (1995)
10. Microsoft Corporation, "Distributed Component Object Model Protocol – DCOM/1.0" (1998)
11. Object Management Group, "Transaction Service", Version 1.2, formal/01-05-02 (2001)
12. Object Management Group, "Concurrency Service", Version 1.0, formal/00-06-14 (2000)
13. Object Management Group, "CORBA Components", Version 3.0, formal/02-06-65 (2002)
14. ObjectWeb, http://www.objectweb.org/
15. ObjectWeb, "The Fractal Composition Framework Specification", Version 1.0, http://www.objectweb.org/fractal/ (2002)
16. ObjectWeb, "The Java Open Transaction Manager", http://jotm.objectweb.org/
17. Plasil, F., Visnovsky, S., "Behavior Protocols for Software Components", IEEE Transactions on Software Engineering, Vol. 28, No. 11 (2002)
18. Popovici, A., Alonso, G., Gross, T., "Spontaneous Container Services", in Proceedings of the 17th European Conference on Object-Oriented Programming, Darmstadt, Germany (2003)
19. Prochazka, M., "Advanced Transactions in Enterprise JavaBeans", in Proceedings of the Engineering Distributed Objects (EDO) Workshop, Davis, USA (2000)
20. 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 (2001)
21. Prochazka, M., "Advanced Transactions in Component-Based Software Architectures", Ph.D. thesis, Charles University, University of Evry (2002)
22. Prochazka, M., "A Flexible Framework for Adding Transactions to Components", the 8th International Workshop on Component-Oriented Programming (WCOP 2003, in conjunction with ECOOP 2003), Darmstadt, Germany (2003)
23. Rouvoy, R., Merle, P., "Abstraction of Transaction Demarcation in Component-Oriented Platforms", ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil (2003)
24. Sun Microsystems, "Enterprise JavaBeans Specification", Version 2.0, Final Release (2001)
25. Sun Microsystems, "Java Transaction API (JTA)", Version 1.01 (1999)
26. X/Open Distributed Transaction Processing: Reference Model, Version 3 (1996)