

Transaction Models vers. Behavior Protocols

Marek Prochazka¹, Frantisek Plasil^{1,2}

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

²Academy of Sciences of the Czech
Republic
Institute of Computer Science
Pod vodarenskou vezi 2
180 00 Prague 8
Czech Republic
{plasil}@uivt.cas.cz
<http://www.uivt.cas.cz>

Abstract

In this paper, we propose a method of transactional behavior description in component-based software architectures. We focus on the specification of transactional models employed in components. Especially, we try to specify the creation and canceling dependencies among transactions. For this purpose, we enhance the SOFA software architecture so that it can be dynamic: at runtime, new interface instances can be added or removed, and new components instantiated and tied. Thus, SOFA behavior protocols in such a dynamically reconfiguring architecture are also dynamic. The transaction manager, as the coordinating component, provides an interface for a transaction initiation and completion. If a subtransaction of a transaction is created, a new transactional interface instance is created and a protocol of the parent transaction is modified. The key achievement of the paper is that it introduces a method for semi-formal specification of the transactional model used by a particular component (replacing thus the classical transactional specification in plain English).

1 Introduction

It is broadly accepted that in the near future software will be composed of small parts with well defined interfaces - software components. Recently, we enhanced our SOFA component model by behavior protocols which extend a component interface with component behavior specification [6]. An interesting area of behavior protocols employment is a component's participation in transactions. For example, as there are many transactional models a transaction manager as the cooperating component should specify which of the transactional models it supports. Naturally, specifying the particular transactional model in plain English is not sufficient for this purpose.

The goal of the paper is to demonstrate the expressive power of behavior protocols in describing components' transactional behavior, especially the ability to clearly reflect the transactional model chosen. In principle, in an application, the architecture protocol of the application's transaction manager determines the current transaction model supported by the transactional manager. The transaction model is not hard-wired, it is specified by *protocol modification rules*.

The paper is organized as follows. In Section 2 we provide an overview of behavior protocols. Section 3 shows how transactional models are specified using behavior protocols. Moreover, dynamic behavior protocols are introduced in this section. In Section 4, a discussion of and a comparison to other approaches are provided. Section 5 is devoted to a brief discussion of future inventions. Section 6 concludes the paper by summarizing the key achievements.

2 Component behavior specification

2.1 Component model

In SOFA ([2],[3]), an application is built as a hierarchy of software components. Analogously with the concept of an object as an instance of a class, we introduce *software component* as an instance of *component template*. The *template frame* defines a set of individual interfaces that are *provided* or *required* by a template. Interfaces, frames and architectures built from frame instances are defined in the SOFA CDL architecture description language [5]. In an architecture, there are three types of interface ties. (1) Binding of a requires-interface to a provides-interface. (2) Delegating from a provides-interface to a nested component's provides-interface. (3) Subsuming of a subcomponent's require-interface to a require-interface.

2.2 Behavior protocols

The basic idea of behavior protocols is to consider a communication between components through interfaces as sequences of communication events. An event can be a *request* or a *response*; we distinguish those events by suffix \uparrow resp. \downarrow . To express, whether an event is emitted or absorbed, we prefix the event identification by the $!$ resp. $?$ symbol. *Behavior protocol* (*protocol* for short) is a regular-like expression, which specifies the required ordering of method calls (requests and responses). In a behavior protocol, a sequence, alternative, repetition, a reentrant or a parallel processing of a method (subprotocol in general) can be specified [6].

2.3 Interface protocol, frame protocol, architecture protocol

In SOFA CDL, behavior protocols are associated with interface, frame, and architecture specifications [6]:

An *Interface protocol* specifies the way of using a particular interface. For example, imagine an Account interface with the `create`, `terminate`, `balance`, `deposit`, and `credit` methods. Such an interface can have the following protocol:

```
interface Account {
    // methods' declaration should be here
    protocol
    create ;
    ( balance + deposit + withdraw )* ;
    terminate
}
```

The protocol specifies that an account is to be created first. Then, the balance can be examined and money can be deposited or withdrawn to/from the account. This can be repeated many times. Finally, the account can be terminated.

A *frame protocol* specifies dependencies between the provide-interfaces' protocols and requires-interfaces' protocols of a particular frame. As an example, consider the frame `MyAccount` providing the `account` interface:

```
frame MyAccount {
    provides:
        Account a;
    requires:
        Database db;
    protocol:
        ?a.create { !db.addRecord } ;
        ( ?a.balance { !db.query } )
```

```

+ ?a.deposit { !db.modify }
+ ?a.withdraw { !db.modify }
)* ;
?a.terminate { !db.removeRecord }

```

Note that { resp. } symbols postfixing a method m substitute corresponding $m\uparrow$ and $m\downarrow$ events, i.e. $m\{x\} = m\uparrow ; x ; m\downarrow$.

An *architecture protocol* is generated automatically from all frame protocols of the nested frames in the corresponding architecture. It is automatically generated by an CDL compiler; the composition operator is employed in the way that all participated frame protocols "run" in parallel, being synchronized in the moments of mutual communication via their tied interfaces [6].

3 Specifying transactional behavior of components

3.1 Motivation

As for transactional behavior specification, most of the component technologies use a very limited approach based on plain-English description. As an aside, they are usually based on an implicitly chosen simple transactional model - usually a flat or nested one. Briefly, a transactional behavior is determined by: (1) the way transactional context is propagated (involves creating, suspending, passing, and deleting a context), (2) the commit protocols policy and (3) the transactional model chosen.

The current technologies include CORBA Object Transaction Service (OTS), Enterprise JavaBeans (EJB), or Microsoft Transaction Server (MTS):

- CORBA Object Transaction Service [7] uses the nested transactional model. Technically, on the ORB, the requests targeting a server object which is subject to a transaction (transactional object) contain a record with transactional context bearing all the necessary information about the current transaction. Conceptually, transactional context is propagated implicitly (involves all the methods of a particular interface) or explicitly. An explicit propagation gives the possibility to determine the context propagation with the granularity of single methods thus making it possible to emulate creating suspending, and passing a transactional context.

Since transactions are coordinated directly by the ORB which, therefore, takes the role of a transaction manager, the transactional functionality of CORBA OTS is wired-in inside the ORB.

- Enterprise JavaBeans technology [22] supports the flat transactional model. An enterprise bean is deployed to EJB container together with its deployment descriptor (this could be considered an ADL level), which determines, via transactional attributes of particular bean methods, the semantics of transaction context propagation. However, only some combination of creating, suspending, passing can be specified this way. For example, a particular method of a bean can be associated with the TX_REQUIRES attribute. In this case, the method is always invoked in the scope of a transaction in the following sense: If the method is not invoked in the scope of a client's transaction, the EJB container creates a new transaction and the method is called in the scope of the new created transaction.

- Microsoft Transaction Server [23] uses transactional attributes with similar functionality for specifying transactional behavior of COM component methods.

To summarize, all these technologies use a hard-wired transactional model, the way of context propagation is alternatively determined by the underlying implementation or can be chosen from predefined options (important special cases are predefined by the technology in question), and the commit protocols are also predefined by the particular technology (all of these technologies support the two-phase protocol with actions). By all means, this situation cannot be considered satisfactory particularly with respect to the fact that conceptually most of the component semantics should be expressed at the ADL level. It is the key goal

of this paper to target this issue.

3.2 Basic idea

Our key idea is to specify component transactional behavior via behavior protocols. In principle, we will assume that, in a particular component based application, there is a transaction manager component that coordinates transactions. Such a transaction manager provides the following Transaction interface:

```
interface Transaction
    //specification of TrBegin, TrCommit, and TrAbort becomes here
    protocol
        TrBegin ; ( TrCommit + TrAbort )
```

A transaction manager is usually a multithreaded server, because several transactions are typically to be processed concurrently. In principle, to express multithreading in protocols, we could use the reentrancy operator (^) as described in [6]. However, we need to capture the notion of "session" which is not directly supported by the ^ operator. Moreover, in some frame specifications it is desirable to express dependencies among client threads, which is also not possible via ^. Therefore, we introduce *dynamic interface instances*. A frame can provide (or require) a pool of instances of the same interface to express that methods are implicitly parametrized by the established session. This parametrization is expressed via the index of an interface instance.

3.3 Reflecting a particular transactional model

In this section, we introduce our approach to transactional models specification by means of behavior protocols. In principle, a transactional model is characterized by: (1) the participating (sub)transactions' control flow, usually specified in terms of dependencies between transactions, and (2) the way of supporting consistency, visibility, and recovery (in other words by the transactions' effects on data objects, i.e., resources).

In this paper, we limit ourselves to reflection of transactional models in terms of control flow. We will show that a state diagram of a transaction can be approximated by a behavior protocol. This has the advantage that we do not have to know any details of the internals of the corresponding transaction manager. Instead, the knowledge of the interface and frame protocols of the transaction manager characterizes the transactional model which the transaction manager implements.

Let us look at the nested transaction model more closely. The transaction manager TM supporting nested transaction provides one instance of the `NestedTransaction` interface representing the root transaction:

```
interface INestedTransaction
    // methods begin, split, commit, and abort to be specified here
    protocol
        begin ; split* ; ( commit + abort )
```

Methods of this interface corresponds to the significant events of the nested transaction model. If a client is bound to this interface, as a response to its invocation of `split`, a new `NestedTransaction` interface instance is created in TM. The TM frame protocol has to be modified, because there are dependencies between the two `NestedTransaction` interface instances: when `commit` is invoked in the root transaction, TM has to wait until the child transaction has finished; when `abort` is invoked in the root transaction, the `abort` method of the child transaction has to be invoked by TM.

An example of a transaction manager architecture supporting nested transactions is on Figure 1. To support creation of nested transactions, there is a *transaction factory* component providing the `ITrFactory` interface. The `ITrFactory` interface supports the creation of an initial transaction (the root nested transaction in our case); the interface protocol takes the form:

```

interface ITrFactory
...
protocol
  ( InitNestedTransaction )*

```

In general, a transaction manager can support many transaction models; for example, ITrFactory can create an initial saga transaction, or a split transaction. There is the TransactionFactory frame (TF in Figure 1) providing ITrFactory interface. In our example, the TransactionFactory frame supports creation of nested transactions. The TransactionFactory frame provides ITrFactory interface only; therefore, its frame protocol does not differ from the ITrFactory interface protocol :

```

frame TransactionFactory {
  provides:
    ITrFactory iTrFact;
  protocol:
    ( ?iTrFact.InitNestedTransaction )*
}

```

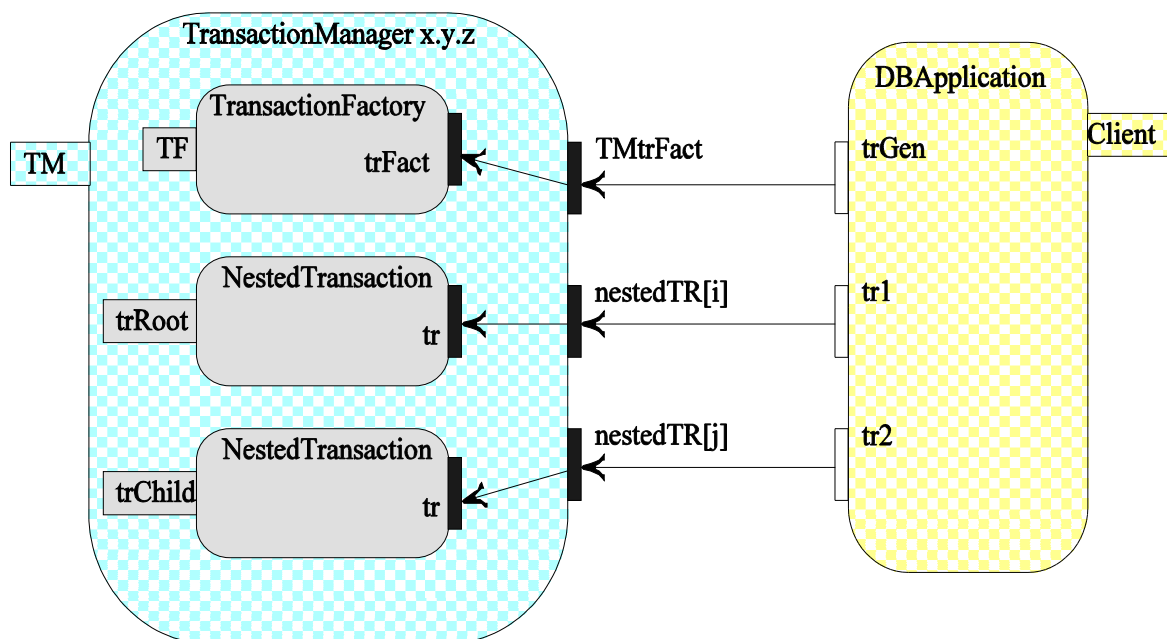


Figure 1: Architecture of a transaction manager supporting nested transactions

The NestedTransaction frame provides the INestedTransaction interface. When the split method is invoked, a new subtransaction is created. Each transaction may have many subtransactions. The NestedTransaction frame provides the INestedTransaction interface only, so its protocol doesn't differ from the INestedTransaction interface protocol:

```

frame NestedTransaction {
  provides
    INestedTransaction iTr;
  protocol:
    ?iTr.begin ; ?iTr.split* ; ( ?iTr.commit + ?iTr.abort )
}

```

3.4 Modeling dependencies using protocols: dynamic protocols

As mentioned in Section 3.2, we intend to introduce dynamic behavior protocols. In fact, the main obstacle of employing the current version of SOFA behavior protocols is their static character. They are constructed to specify the behavior of a static software architecture. Not considering the (a bit clumsy) reentrancy operator \wedge , even in one of the simplest transactional models - nested transactional model, we can not specify the control flow of a transaction behavior protocols in their form specified in [6]. The reason is that, in general, we do not know the structure of the transaction (the tree structure of nested transactions) in advance. A proposed remedy is to allow dynamic modification of behavior protocols, e.g., by specifying some rules for "well-done" modifications.

3.4.1 Transaction manager frame protocol

The TransactionManager frame provides one instance of the ITrFactory interface and a pool of the INestedTransaction interface instances. The new event and the rules section are introduced to achieve the required dynamics in the TransactionManager frame protocol:

```
frame TransactionManager {
  provides:
    ITrFactory iTMTrFact;
    INestedTransaction iNestedTr[];
  protocol:
    ( ?iTMTrFact.InitNestedTransaction { new INestedTransaction iRootTr } ;
      ?iRootTr.begin ;
      ?iRootTr.split* ;
      ( ?iRootTr.commit + ?iRootTr.abort )
    )^
  rules:
    INestedTransaction iTrParent.split {
      new INestedTransaction iTrChild ;
      ( ?iTrParent.commit! ->
        ?iTrParent.commit! ; ( ??iTrChild.commit + ??iTrChild.abort )
      )
      ( ?iTrParent.abort! ->
        ?iTrParent.abort! ; !!iTrChild.abort
      )
    }
}
```

The new event specifies creating a new interface instance in the pool of interface instances. The name of this newly created interface instance can be used later in the protocol. This name is not a real instance name. Note that the new event can be applied to those interfaces, which are pooled in a frame (this is specified using [] brackets in the provides section). While events on static interfaces can be used in the correspond frame protocols, events on dynamically created interfaces cannot be used. Newly created interface's protocol is used as a new part of the frame protocol; this new part joined together with the original protocol using composition operator in the following way:

```
old_protocol  $\sqcap$  new_part_of_protocol
```

The rules keyword begins the *frame protocol modification section* specifying frame protocol modifications done after the appearance of specified events. To allow to change a particular frame protocol after an event, *protocol modification rules* are introduced. Each rule has a left side and a right side separated by the \rightarrow operator. On the left side, there is a part of a frame protocol, which is replaced with the protocol on the right side of the rule. Moreover, to allow to synchronize parts of a frame protocols joined by the composition operator, new primitives are added to the protocol. Using ?? and !! prefixes you can synchronize any two methods within protocol parts joined by the composition. ??inst.method in protocol specifies that the next accepted event will be the method invocation on the inst interface instance. !!inst.method event meaning is that the method method is invoked on inst interface instance.

In our example, the parent transaction has to wait until the child transaction is finished (the parent transaction is abort dependent on the child). When the parent transaction is aborted, the child transaction is also aborted (the child transaction is abort dependent on the parent).

Before the creation of a subtransaction of the root transaction, a transaction manager frame protocol looks as defined in the protocol section. After the creation of a subtransaction, in other words, after the application of `iTrParent.split` event rule, the transaction manager frame protocol looks as follows:

```
( ?iTMTrFact.InitNestedTransaction { new INestedTransaction iRootTr } ;
  ?iRootTr.begin ;
  ?iRootTr.split* ;
  ( ?iRootTr.commit + ?iRootTr.abort )
)^
□
( ?iChildTr.begin ;
  ?iChildTr.split* ;
  ( iChildTr.commit {
    ??iRootTr.commit
    + ??iRootTr.abort
  } +
  ?iChildTr.abort {
    !!iRootTr.abort
  }
  )
)
```

Similarly, after the creation of a new subtransaction of any existing transaction, a new part of the frame protocol will be created and joined with the original part using composition and new dependencies by means of protocol parts' synchronizations will be established. Note that we also introduce the `delete` event for the deleting interface instances and the corresponding synchronization events' removal.

3.4.2 Transaction manager architecture protocol

A transaction manager architecture depicted at Figure 1 is specified using the following protocol:

```
architecture TransactionManager version x.y.z {
  inst TransactionFactory TF;
  inst NestedTransaction NestedTr[];
  delegate iTMTrFact to TF:iTrFact;
  rules:
    ?TF.InitNestedTransaction {
      new NestedTransaction TrRoot ;
      new INestedTransaction iTrRoot ;
      delegate iTrRoot to TrRoot:iTr ;
    }
    +
    ?NestedTransaction TrParent.split {
      new NestedTransaction TrChild ;
      new InestedTransaction iTrChild ;
      delegate iTrChild to TrChild:iTr ;
    }
}
```

Transaction manager supporting nested transactions compounds one instance of the `TransactionFactory` frame and many instances of the `NestedTransaction` frame. Each instance represents one subtransaction (considering that root transaction is also subtransaction). There is one news in the architecture description: the `rules` section, which begins the *architecture modification section* specifying architecture protocol modifications done after the appearance of specified events. Here, new instances can be created (using the `new` event), deleted (using the `delete` event), tied (using the `bind`, `delegate`, and `subsume` events), and untied (using the `disconnect` event).

The real picture of a transaction manger is, of course, more complicated. There are more special components

involved to transactional software system; log manager, lock manager, and resources are some of them.

3.5 Modeling permit and delegate using protocols

In the previous section, we focused on transactions' flow control. We omitted what visibility, recovery and consistency address: There are also data objects involved to transactions; each data object is represented via a frame and each object has its frame protocol. In our idea, delegation of resources can be viewed as moving parts of protocols from one frame to another. Similarly, giving permissions to invoke some operations can be viewed as publishing parts of protocol to another frame. These techniques are one of our nearest future intentions. We have to design *transactional environment*, where resources are registered to transactions, delegated from one transaction to another, and permitted for using by another transaction.

4 Discussion, related work

The proposed method makes it possible to specify transactional model supported by a transaction manager. Dependencies among transactions are not described in plain English; instead, they are specified by dynamically modifiable regular-like expressions. A particular transaction model is characterized by the initial transaction manager's protocol and by a set of protocol modification rules. Our approach allows for a specification of any (potentially even user-defined) transactional model. Even though we use the SOFA notation in this paper, but our approach is applicable in component-based architectures in general.

Chrysanthis and Ramamrithan in [9], [10], [11], and [12] introduce ACTA, a formal framework for specifying extended transaction models. ACTA allows for intuitive and precise specification of extended transaction models by characterizing the semantics of interactions between transactions in terms of different dependencies between transactions, and in terms of transaction's effects on data objects. However, ACTA framework is not focused on components exhibiting their behavior. Moreover, behavior protocols make it possible to check whether a particular transaction model is abided.

Aster [8] extends its architecture description language by non-functional properties defined by means of temporal logic rules. Transaction models are defined by such rules and an architecture is build from middleware components associated with the required non-functional properties using integration rules. However, the integration rules approach does not scale well if more middleware components are used to describe the architecture. Moreover, the specification of transaction models is not easy to read..

ASSET [13] provides a set of transaction primitives extending a programming language. Using dependencies, delegation and permitting, a programmer can create any transaction model. In contrast to our approach, the transactional behavior is not revealed in an explicit specification; instead, the transaction model is specified indirectly, being spread over a programming language code.

PJama [24], [25] extends an orthogonal persistent language by support for customizable transactions. However, the current version of PJama does not support extended transaction models.

5 Future intentions

In future, we would like to specify transaction models by means of behavior protocols more precisely. Also, we intend to design a transactional environment, where transactions could be mutually dependent, data resources could be registered to a transaction, delegated from a transaction to another transaction, and permissions to access resources could be delegated by a transaction to another transactions. We intend to elaborate on the idea that delegation of resources can be viewed as moving parts of protocols from one frame to another. Similarly, giving permissions to invoke some operations could be viewed as transferring parts of protocol to another frame. We would like to create a prototype of transactional SOFA node, which will participate in transactions and which will reflect the transactional behavior of a transaction manager.

6 Conclusion

This paper proposes a method for describing transactional behavior in component-based software architectures. The key idea is the specification of transactional models by means of behavior protocols, which especially includes the specification of dependencies among transactions. The SOFA component model is extended in the way that it is dynamic: New interface instances or ties to other components can be created or canceled at runtime. Therefore, behavior protocols of these dynamic architectures are also of dynamic nature. Transaction manager as a common component provides an interface for a transaction initiation and completion. If a subtransaction of a transaction is created, a new transactional interface instance is created and a protocol of the original transaction is modified. The key achievement of the paper is that it introduces a method for semi-formal specification of transactional model used by a particular component, instead of a specification in plain English. An architecture protocol of transaction manager determines a corresponding transaction model. A transaction model is not hard wired, it is specified by means of protocol modification rules.

References

- [1] Plasil, F., Mikusik, D.: Inheriting Synchronization Protocols via Sound Enrichment Rules. In Proceedings of Joint Modular Programming Languages Conference, Springer LNCS 1204, March 1997.
- [2] Plasil, F., Balek, D., Janecek, R., Pospisil, R., Prochazka, M.: SOFAnet and SOFAnode – Basic Functionality. TR 97/12, Dept. of Software Engineering, Charles University, Prague, 1997.
- [3] Plasil, F., Balek, D., Janecek, R.: SOFA/DCUP Architecture for Component Trading and Dynamic Updating. In Proceedings of ICCDS '98, Annapolis, IEEE CS, 1998, pp. 43–52.
- [4] Mencl, V.: Component Definition Language, Master thesis, Charles University, Prague, 1998.
- [5] Mikusik, D., Stranik, J., Svec, M., Visnovsky, S.: Synchronization protocols for Orbix 2.0, Dept. of Software Engineering, Charles University, Prague, 1998, <http://nenya.ms.mff.cuni.cz/~svec>.
- [6] Plasil, F., Visnovsky, S., Besta, M.: Bounding Component Behavior via Protocols. Accepted to TOOLS USA '99 Conference, August, 1999
- [7] Object Management Group: Object Transaction Service. December, 1997.
- [8] Zarras, A., Issarny, V.: A Framework for Systematic Synthesis of Transactional Middleware. Proceedings of Middleware '98, September 1998.
- [9] Chrysanthis, P. K., Ramamritham, K.: A Unifying Framework for Transactions in Competitive and Cooperative Environments. IEEE Office and Knowledge Engineering, 4(1):3-22, February 1991.
- [10] Chrysanthis, P. K.: ACTA, A Framework for Modeling and Reasoning about Extended Transactions Models. Ph.D. Thesis, September 1991.
- [11] Chrysanthis, P. K., Ramamritham, K.: Synthesis of Extended Transaction Models Using ACTA. ACM Transactions on Database Systems, 19(3): 450-491, September 1994.
- [12] Chrysanthis, P. K., Ramamritham, K.: ACTA: a framework for specifying and reasoning about transaction structure and behavior. Readings in Database Systems, 2nd ed., M. Stonebraker, Ed., Morgan Kaufmann, pp. 335-344, 1994.
- [13] Biliris, A., Dar, S., Gehani, N., Jagadish, H.V., Ramamritham, K.: ASSET: A System for Supporting Extended Transactions. Proceedings of ACM SIGMOD International Conference on Management of Data, May 1994
- [14] Mohan, C.: Advanced Transaction Models: Survey and Critique. ACM Sigmod, International Conference on Management of Data, May 1994.
- [15] Wachter, H., Reuter, A.: The ConTract Model. In Ahmed K. Elmagarmid: Database Transaction Models for Advanced Applications, 1991.
- [16] Georgakopoulos, D., Hornick, M. F., Manola, F., Brodie, M. L., Heiler, S., Nayeri, F., Hurwitz, B.: An Extended Transactional Environment for Workflows in Distributed Object Computing. IEEE Bulletin of the Technical Committee on Data Engineering, June 1993.
- [17] Bukhres, O., Elmagarmid, A., Kuhn, E.: Implementation of the Flex Transaction Model. IEEE Bulletin of the Technical Committee on Data Engineering, June 1993.
- [18] Lomet, D., editor: Special Issue on Workflow and Extended Transaction Systems. IEEE Bulletin of the Technical Committee

on Data Engineering, June 1993.

- [19] Shrivastava, S. K., Wheeler, S. M.: Implementing Fault-Tolerant Distributed Applications Using Objects and Multi-Coloured Actions. 10th International Conference on Distributed Systems, Paris, May 1990.
- [20] Jajodia, S., Kerchsberg, L.: Advanced Transaction Models And Architectures. Kluwer, 1997.
- [21] Elmagarmid, A. K.: Database Transaction Models For Advanced Applications. Morgan Kaufmann, 1992.
- [22] Matena, V., Hapner, M.: Enterprise Java Beans Specification 1.0. JavaSoft, March 1998.
- [23] Jennings, R.: Microsoft Transaction Server 2.0. Database workshop, Sams Publishing, 1997.
- [24] Daynes, L., et al.: Customizable Concurrency Control for Persistent Java. Chapter Seven in Advanced Transaction Models and Architectures, Editors: S. Jajodia & L. Kerschberg, August 1997.
- [25] Daynes, L.: Extensible Transaction Management in PJava. First International Workshop on Persistence and Java, September 1996.
- [26] Wu, Z., Schwiderski, S.: Reflective Java: Making Java Even More Flexible, February 1997.
- [27] Heineman, G. T., Kaiser, G. E.: The CORD approach to Extensible Concurrency Control, IEEE International Conference on Data Engineering, 1997.