ReflecTS; A Reflective Transaction Service Framework for Open Applications

Anna-Brith A. Jakobsen and Randi Karlsen
{annab,randi}@cs.uit.no

Computer science department, University of Tromsø, 9037 Tromsø, Norway

Abstract. Transactional middleware platforms must accommodate an increasingly diverse range of requirements from both applications and the underlying systems. It is clear that applications have characteristics and requirements that vary a lot, and that transactional middleware must be able to support the potential variety in transaction execution requirements. In this paper we describe ReflecTS, a reflective platform for transaction services that will meet the diverse needs of applications. The platform, which is composed of components and component frameworks, supports concurrently running transaction services and exposes the ability to configure and reconfigure the services by adopting the principles of reflection.

1 Introduction

Middleware has evolved as a standard way to implement distributed applications. Middleware infrastructures, for instance CORBA [22], Java RMI [35] and DCOM / .Net [15], represents a software layer between the application and the underlying operating system, hiding distribution and heterogeneity for the above layer. *Transactional middleware* supports the execution of distributed transactions running on heterogeneous, distributed hosts. Present implementations of transactional middleware, like CORBA's Object Transaction Service (OTS)[2] and Enterprise JavaBeans' Java Transaction Service (JTS) [43], provides mainly support for the traditional flat transaction model while preserving the *ACID* (atomicity, consistency, isolation and durability) properties.

The traditional flat transaction concept is generally not applicable within advanced applications like workflow, cooperating work, multimedia and mobile applications, where characteristics and transactional requirements goes beyond the ACID properties. We are in this paper particulary concerned with applications that goes even a step further and exhibits characteristics and properties that can vary over time or be dependent on the situation the applications are used in. Such applications appears to us as *open application domains*. Open application domains are characterized by unpredictability and major variations in transactional requirements, which also may lead to new and demanding requirements.

To oblige unpredictability and requirements from open application domains, we propose in this paper a highly adaptable and flexible transactional middleware platform. The platform will meet the varying transactional requirements by offering a number of *concurrently running transaction services* each providing different transactional guarantees, and by applying qualities to *configure and reconfigure* transaction services.

We design the platform using *components*, *component frameworks* and *reflection*, which are the general means to achieve configurability and reconfigurability within middleware today. The platform will be composed of components throughout the whole platform. Domain-specific component frameworks will constrain the design space and the scope for evolution of the platform. By deploying an extensible set of transaction services within the framework, we argue that our approach considerably will improve the support of varying transactional requirements of applications.

This work is a part of the Arctic Beans project[5], which is funded by The Research Council of Norway. The primary goal of the Arctic Beans project is to provide a more open and flexible enterprise component technology, with support for configurability and re-configurability, based on the principles of reflection.

In the remainder of this paper we first, in section 2, discuss the motivation for this work. Section 3 will give necessary background information on global transaction processing, components, component frameworks and reflection. Then a specification and a overall design of the flexible transactional middleware platform follows in section 4. Section 5 gives presents an architecture for an implementation and lists issues that follows in the wake of an implementation. Section 6 presents related work. Finally, section 7 draws conclusions and presents current and future work.

2 Motivation

In traditional database systems, the transaction concept is related to the flat transaction model and the key to success is the traditional ACID properties. However, new applications have varying characteristics and transactional requirements, and the traditional transaction model is generally not applicable. Many advanced applications, within for instance workflow, mobile, multimedia and cooperating application domain, execute long-running transactions where strict ACID properties are not particularly suitable. Strict atomicity would, for long-running transactions, imply too much work to be undone in case of failure, and strict isolation prevents concurrency and co-operation.

To meet requirements from advanced applications, a number of advanced transaction models [18, 39] have been proposed during the last decades. They address specific transactional requirements, such as relaxed atomicity and isolation requirements, and offers thereby some flexibility. They do, however, not support the required flexibility in a principled way, and are neither implemented nor used in any commercial product.

Advanced applications embrace a huge number of very different types. For some of them we see a need for adaptivity, as the applications exhibits characteristics and properties that vary over time and from situation to situation. Such *open applications* may need to execute transactions with different properties, and the transaction management system must adapt accordingly. Some open applications initially requires only a single set of transactional properties. This set may change during runtime, for instance from the ACID properties to properties where semantic atomicity, relaxed isolation and / or timely constraints are included. Other open applications may concurrently execute transactions that require different properties. We assume that open applications are under constant evolution and that new requirements may emerge over time. Transactional requirements are thus not fully known at application design time.

A medical information systems is an example of an application with dynamically changing transactional requirements. Within these systems, information of different types; for instance patient journals, radiographs and spoken reports are stored over a number of sites. A variety of different applications may work towards these sites. For instance, user A works with updates of patient journals, and initiates the execution of traditional flat transactions which requires the ACID properties. User B works with statistical data, and preferably initiates long-running transactions that will scan a number of patient journal databases searching for specific information. This transaction may not follow the traditional ACID properties as statistical data can be indicated truth rather than exact truth. User C wants to access a centrally stored patient journal while driving to a patient, so transaction processing in this setting must take into account mobility issues as well as transactional guarantees. User D is on a place of loss needing important information from a patient journal. User D does not have much time available, so he will typically ask for immediate response on his request. User A and user D do issue the same kind of transaction; reading and possibly updating information from a patient journal. However, they have different expectations and requirements to the transaction. User E, also on a place of loss, is equipped with a PDA and sensor- and recording devices. User E communicates with for instance a hospital to where he wants to transfer multimedia information in real-time. Such transactions requires quality of service guarantees.

The picture drawn above can be more complex and the transactions more structured. For instance, compensating transactions may be necessary, and sub-transactions can be specified as optional, contingent, with retry options, or with dependency on the outcome of other transactions. What we do have seen is that different user groups and different applications can issue different types of transactions, from simple to complex. This literally means that the different types of transactions can be described using different transaction models, either models that are already described [18, 39][17][3], or models that can be described.

Even though different transaction models exist, none of them alone offer the required flexibility in open applications. We therefore argue that transactional middleware must support the varying needs from open applications by providing a transactional framework where a number of *concurrently running transaction services* can serve the different transactions.

At application design time, there may be a need to decide upon a transaction service for the application. If the required transaction service does not exist within the pool of available transaction services, the application designer may want to *configuration* a suitable transaction service for the particular application. An application can change its needs during run-time. It may either be that new needs arises, or that the application designer where not aware of all transactional needs at initialization time. This initiate a *reconfiguration* of an existing transaction service or a *configuration* of a new service. Due to this we argue that a flexible transactional middleware also must expose the ability to configure and reconfigure transaction services.

3 Background

Global transaction processing has been an issue in multidatabase systems for many years. As the requirements to transaction processing has evolved from being static to variable and flexible, building transaction processing systems requires technology that can oblige them. In the following, we first give an overview of global transaction processing, and thereafter we give information about technology to support the development of flexible transaction processing systems.

3.1 Global Transaction Processing

Most advanced applications today requires access to a number of various distributed and heterogeneous data sources; for example a database, a file or a printer, which are managed by resource managers (RM). Systems that facilitate the local integration of local data sources are called multidatabase systems (MDBS) [10]. A MDBS is a layered architecture built on top of a number of resource managers (RMs). Access to data in local data sources are accomplished through for instance transactions. A MDBS supports two types of transactions; 1) Local transactions are executed by the local RMs, outside of MDBS control; 2) Global transactions are executed under MDBS control. A global transaction consists of a number of subtransactions, each of which is an ordinary local transaction executed at a local DBMS. Within a MDBS system, a *global transaction manager (GTM or only TM)* coordinates distributed transaction processing and submits global transaction operations to the local RMs.

Figure 1 depicts an overview of global transaction processing within a traditional multidatabase system.

3.2 Reflection, Components and Component Frameworks

Reflection is based on the idea of open implementation [32], which is about designing reusable software modules. Reflection is the principled means to achieve open implementations as reflection is a technique applied for “opening up” a system to support inspection and adaptation of internal structure and behaviour [34][31]. Aspects of the system are available through

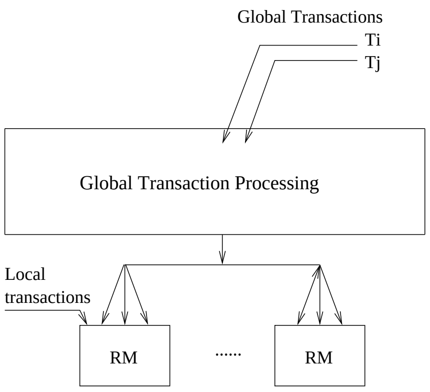

Fig. 1. Global Transaction Processing

offered meta-interfaces (or meta-object protocols, MOP). A meta-interface will typically provide operations to inspect the internal details of the platform (introspection) and to change the underlying middleware (adaptation). More generally, there are two styles of reflection : *Structural* reflection to inspect and change the underlying structure of the system, and *behavioural* reflection to inspect and change the activity in the underlying system [7].

Component technology has come to be the most widely adopted technique to construct configurable software systems. According to Szyperski [44], a component can be viewed as:

> *"a unit of composition with contractually specified interfaces and explicit context dependencies, and in this context, a component can be deployed independently and is subject to third-party composition".*

Components adds *flexibility* to software systems. Components implement strong interfaces and encapsulates implementation details, which enable systems to be easily adapted by adding, removing or replacing components. A number of component models, both commercial and research based, are available. Enterprise JavaBeans [36] and CORBA Component Model [23] are heavy-weight enterprise components. Java Beans [35] has provided support for reflection (introspection and specialization). Microsoft's Component Object Model (COM) [14] is a model performing more efficiently than Java Beans, but provides no reflective capabilities. The .NET [15] framework from Microsoft provides another component model where reflective capabilities are fully available to introspect metadata and components. OpenCOM [13] is a lightweight, efficient and reflective component model built upon the core concepts of COM (i.e. uniquely specified and discoverable interfaces), omitting higher-level features such as distribution, persistence, transactions and security. Reflective features for introspection and adaptation are added to OpenCOM, making it an ideal component model to build reflective middleware upon. OpenCOM was designed specifically for the implementation of the OpenORB reflective middleware [9].

An open and reflective middleware infrastructure build upon components, are subject to dynamic changes. The importance of enforcing architectural constraints on a dynamically evolving platform is considerably, as we require integrity to be preserved at any time. To achieve integrity preservation, the *component framework* (CF) technology is applied. A CF is defined by Szyperski [44] as:

> *"a collection of rules and interfaces that govern the interaction of a set of components plugged into them".*

A component framework enforces architectural principles on the components it supports, preserving integrity and constraining the design space and the scope of evolution. Component frameworks applies to specific domains, which means that integrity maintenance are closely related to the characteristics of the domain. For example, a *transaction service component* *framework* will contain specific rules for the architecture of a transaction service. A component framework will also simplify the assembling of components. To do this, it maintains an architecture consisting of a component graph and its constraints.

There are few component frameworks available for commercial use. OpenORB[9] and ReMMoC [20] are implemented using OpenCOM components, and creates their own component framework model atop OpenCOM. Other component frameworks are OpenDOC [27] and BlackBox [28].

4 Design of ReflecTS; A Reflective Transactional Middleware Platform

4.1 Introduction

This section describes our current activities in order to meet the goals described introductorily. While transactional middleware like OTS and JTS assures the ACID properties for distributed transactions, our work is designed to meet varying transactional requirements imposed by open applications. We design an architecture called *ReflecTS* (**Reflec**tive **T**ransaction **S**ervice middleware platform). ReflecTS will provide support in the deficiencies identified within current transactional middleware by adopting the following approaches:

1. ReflecTS will provide both initial and run-time *configuration* of transaction services
2. ReflecTS will provide run-time *reconfiguration* of transaction services
3. ReflecTS will offer *concurrently running transaction services* guaranteeing a variety of transactional requirements

The underpinning key technologies of ReflecTS are components, component frameworks and reflection. Components, the main building-blocks of the platform, as they are self-contained, reusable, replaceable architectures with strong interfaces; Component frameworks for constraining the scope of evolution, configuration and reconfiguration of the components forming the platform; And reflection as a general mean to open up the system and achieve configurability and reconfigurability.

The base functionality of ReflecTS, the TSenvironment [30, 29] represents a model for an adaptable transactional system and is briefly described below.

4.2 Model for an Adaptable Transactional System

The *Transaction service execution environment (TSenvironment)* [30, 29] is an architecture for a reflective transactional system. Within the TSenvironment, transaction services can be deployed, modified and used concurrently according to the needs of the applications. The TSenvironment includes a *Transaction service manager (TSmanager)* controlling transaction service deployment and modification, and guarantees consistent use of different transaction services. An overview of TSenvironment is given in figure 2.

A *transaction service, TS* is a self-contained software component that is independently developed and delivered to a transactional environment. A TS may be composed of a number of smaller components *(service components)*, which are assembled to form a complete and consistent TS. Service components represent well-defined tasks within a transaction service, for instance commit, recovery and concurrency control.

Component deployment: The TSmanager handles deployment of transaction service components. The TSmanager also stores information about the components as provided in accompanying component descriptors, holding relevant information about the component (e.g. transaction management properties, service composition, conditions for using the component, and component *compatibility*).

Service assembly: A component deployed in the TSenvironment can, as described, either represent a complete TS or a service component that must be assembled with other services to form a complete service. Assembling of service components are based on information from a

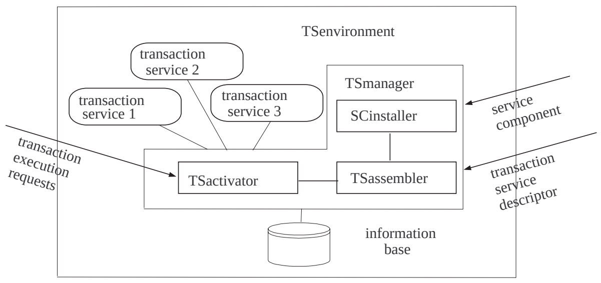

Fig. 2. Overview of TSenvironment

component descriptor. A complete TS is registered in the TSenvironment, and the descriptor stored in an information base.

Service activation: When transaction execution is requested, the TSmanager i) determines the proper TS to use, and ii) allows the TS to start execution when this does not violate correctness.

Based on information from user, available system resources and underlying systems, a suitable TS will be determined from the pool of available services. The selected TS may be either active or inactive. An *active service* is currently managing at least one transaction, while an *inactive service* is currently not used by any transaction. *Compatible* transaction services do not interfere with each other and can be active at the same time. Two services are *incompatible* if the transactional properties of either one of them cannot be guaranteed when they are active at the same time.

4.3 Overview ReflecTS Platform

The ReflecTS platform is composed of components and component frameworks (CF). The platform itself is represented as a CF; *ReflecTS CF*, which contains a number of components; TSActivate, TSInstall and InfoBase, and a CF; *TS Framework*. TS Framework is further configured by plugging in a number of different transaction service implementations. Both complete transaction services (TS) and service components (SC) can be deployed. A complete TS implementation can consist of a number of components, each implementing a welldefined task of the TS. The component frameworks in the platform implements policies for constraining the architecture and the scope of evolution of its constituents. The components implements administrative tasks. ReflecTS CF will provide interfaces to for instance applications and transaction service designers (which can be exposed from plugged in components). An overview of the ReflecTS platform is given in figure 3.

The *TSActivate* component implements an interface through which it will receive transactional requests (i.e. start-transaction) from applications and implement management routines for handling these requests. If an start-transaction request is issued TSActivate selects an appropriate TS and activates it if it is compatible with already active TSs. The transaction will be handed over to the selected TS where the execution of will be controlled.

The *TSInstall* component implements an interface through which it will receive requests for TS and SC deployment and TS configuration and reconfiguration from transaction service designers. Service components or complete transaction services, descriptors and compatibility information will be provided along with the requests. Descriptors and compatibility information will be stored at the InfoBase component before the tasks will be performed and controlled by the TS Framework.

The *InfoBase* component will contain information about deployed transaction services, transaction service descriptors, service component descriptors and transaction service compatibility. A transaction service descriptor contains information about the transactional guarantees the service provides and how to use it. A service component descriptor contains information

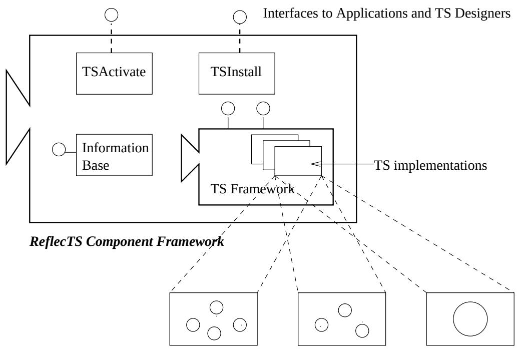
Interfaces to Applications and TS Designers
TSActivate
TSInstall
Information Base
TS Framework
TS implementations
ReflecTS Component Framework

Fig. 3. Overview of the ReflecTS platform

about how a service component can be assembled with other components to form a complete service. Transaction service *compatibility* information, both *vertical* and *horizontal* compatibility is stored as a part of the descriptor. TS's that can be concurrently active without causing any inconsistencies are determined as horizontally compatible. Vertical compatibility is present if transactional protocols (commit, recovery, global concurrency control) in a TS matches the corresponding protocols in the current underlying RMs. More information about transaction service compatibility can be found in section 4.5. Information stored by the InfoBase component will be used both by the TSActivate and the TSInstall component.

The reflective capabilities of the ReflecTS platform is suggested achieved by equipping the CFs with a meta-interface for inspection and reconfiguration as implemented in OpenORB [9] and ReMMoC [21]. We consider two types of reflection, structural and behavioural. Structural reflection on ReflecTS CF involves for instance adding new functionalities to a component (i.e. update information in the InfoBase or replace the TSActivate component) or to add or remove components from the ReflecTS CF. Structural reflection on TS Framework involves configuration and reconfiguration of TS implementations. Behavioural reflection on TS Framework is represented by the ability to select between TS implementations.

The properties of the ReflecTS platform are given by the TS implementations deployed within the platform. Structural reflection on TS Framework will change the properties of the platform. Behavioural reflection will not. Structural reflection on ReflecTS CF will not change the properties of the platform, only change the way the management components behave.

ReflecTS is presented as a self-contained and freestanding solution to transactional middleware. Its ability to be incorporated into a middleware infrastructure providing other functional and non-functional properties, will not be discussed in this paper. An example of ReflecTS in a global setting is presented in figure 4.

4.4 ReflecTS; Design and Responsibilities

This section will give an overview and state the responsibilities of the component frameworks, ReflecTS CF and TS Framework, and the transaction service implementations.

ReflecTS Component Framework The ReflecTS CF conform to the following responsibilities:

1. Act as an access point for transaction service configuration and reconfiguration
2. Act as an access point for transactional requests
3. Provide the ability to change the internals of the ReflecTS CF
4. Assure that configuration and reconfiguration of its plug-ins conforms to a valid architecture

ReflecTS CF meets its first responsibility by exposing an interface provided by the TSInstall component, see figure 3. This interface will be available for transaction service designers, and

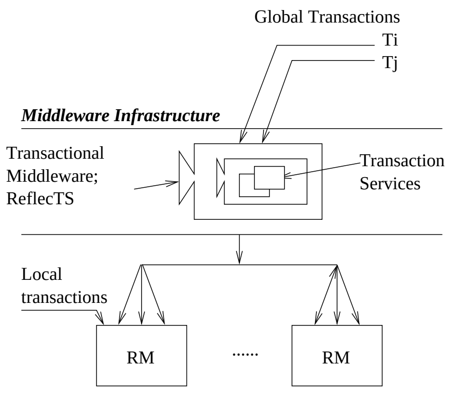

Fig. 4. Transaction Processing within ReflecTS

will contain methods for configuring and reconfiguring TSs. Requests on this interface are managed by the TSInstall component, but are actually performed by the TS Framework where the TSs are deployed.

ReflecTS CF meets responsibility number two by exposing an interface provided by the TSActivate component. This interface will be available for applications and contain methods for starting and committing transactions. Requests on this interface are managed by the TSActivate component before they are actually controlled and performed by a TS within TS Framework.

ReflecTS CF provides the ability to change its internals by implementing a meta-interface with methods for introspection and reconfiguration. Reflection on ReflecTS CF is achieved via this meta-interface, and is basically represented by the ability to change the components or the component structure within it. This is actually performed on a graph of the components, which is maintained by ReflecTS CF.

ReflecTS CF performs integrity maintenance and assures that its constituents conform to a valid architecture. Every change to a transaction service configuration is compared against a set of pre-defined valid architectures. The set of architectures is specific to ReflecTS' domain. It is not static, but can change according to new needs. For example, the architecture of the present version of ReflecTS CF describes how the three specific components, TSActivate, TSInstall and InfoBase, and the component framework, TS Framework, is related to each other. Changes within ReflecTS CF are constrained to match this architecture. If an additional component joins ReflecTS, a new architecture must be described and added to the set of valid architectures.

TS Framework The TS Framework contains an extensible set of transaction service implementations and conform to the following responsibilities:

1. Receive and route transactional requests
2. Provide reflection in order expose the possibility to configure and reconfigure transaction services
3. Assure that configurations and reconfigurations of transaction services conforms to valid services and valid architectures

To respond to its first requirement, TS Framework exposes an interface provided by the deployed TS implementations. This interface contains methods for transactional requests; such as *Trans_Begin()* and *Trans_Commit*. Methods are requested by applications via the TSActivate component. The transactional requests are routed by TS Framework to the correct TS implementation where it is actually performed.

To address the second responsibility, TS Framework provides a meta-interface facilitating the tasks of configuration and reconfiguration of transaction services. This interface contains methods for introspection and reconfiguration of the component graph maintained by the framework. Activity on this interface is initiated by a *Transaction Service Designer*, via the TSInstall component.

To meet responsibility number three, TS Framework implements policies for constraining the scope of configuration and reconfiguration of transaction services. TS Framework maintains a graph of its constituents. When there are changes to the graph, TS Framework assures that the new configuration conforms to a valid architecture. This is done by comparing it against a set of pre-defined valid architectures. A valid architecture also implies a valid service as the architecture constrain not only the number of, but also the type of components (commit, recovery or GCC) that can reside in the graph. New valid architectures can be added to the set when necessary. Besides checking a new structure for validity, TS Framework assures that reconfigurations to a TS are made when the TS is inactive. Otherwise, the results could be compromised or lost.

TS Implementations In general, the responsibility of a TS implementation is to execute transactions according to its guaranteed properties. A TS implementation provides an interface for transactional requests which is exposed by the TS Framework and activated by applications via the TSActivate component.

Traditionally, a transaction service is managed as a single component implementing global commit, recovery and global concurrency control (GCC) mechanisms [45][10]. As an alternative to a 'one-component TS', we suggest composing TS's using a number of components. The main motivation for this is the improved ability to modify the TS by manipulating one its constituents rather than the whole TS. A TS in the ReflecTS platform can consist of one component, or a number of components. Depending on the transactional guarantees provided by a TS and the autonomy of the underlying resource managers (RMs), different versions of global commit, recovery and GCC can be combined. A global commit procedure deals with commit of distributed transactions. Global commit is generally easier when the underlying RM's provide a prepare-to-commit state as specified in the X/Open XA-specification [24]. A recovery procedure basically deals with transactions that are active when a system crashes. The job of a recovery procedure is to bring the databases back to a consistent state. Depending on the atomicity requirement belonging to the transaction, the system will manage to bring the databases completely back to origin. Different recovery procedures uses different methods: different way of logging or checkpoints. GCC deals with assuring serializability and correct behaviour of concurrent transactions using schedulers. This is basically possible when the underlying RM's makes visible its serialization protocol. However, global concurrency control is not always a necessary constituent of a transaction service.

The commit/recovery and GCC procedures must be matching procedures. For instance, a commit protocol supporting an open, nested transaction model where sub-transactions commit before their top-level transaction commit, will match a recovery protocol using compensating transactions during recovery.

TS implementations within TS Framework may for instance consist of global commit, recovery and GCC, or only commit and recovery procedures. We suggest adding a control component to the structure making the component graph of a TS implementing commit, recovery and GCC look like the one in figure 5.

4.5 Transaction Execution Request

Requests for transaction execution are maintained by the TSActivate component before they are eventually executed by a TS implementation. As a start transaction request raises a lot of interesting questions regarding compatibility, it will be examined in the following.

Start_Transaction() Information about the transaction and its transactional requirements is provided along with the request. We assume there exists a programming model for applications, giving the ability to formally specify transactional requirements.

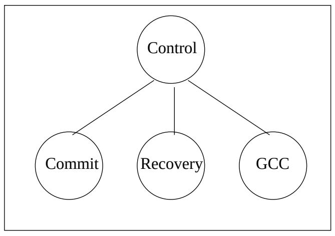

Fig. 5. An example of a TS implementation

The formally specified transactional requirements will be used by a *Selection Procedure* to select a suitable TS. Descriptions about deployed TSs will be gathered from the InfoBase component, and compared with the transactional requirements. A matching TS will be selected to control the current transaction. If a suitable TS cannot be found, the Selection Procedure can for instance implement one of the following options: *1)*Return without a TS, which means that the transaction will not be executed, *2)*Return with a TS that at least provides stronger guarantees than what is required, *3)*Negotiate with either the application designer or the user in order to agree on alternative transactional requirements so that one of the available TSs can be suitable. Figure 6 gives an overview of the selection procedure when option 3) is chosen.

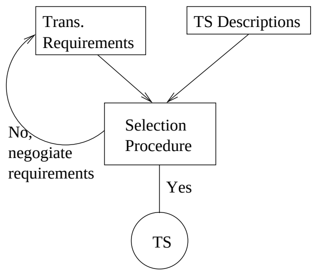

Fig. 6. Overview of the Selection Procedure

After selecting a TS, its compatibility with transactional procedures in involved resource managers (databases, file systems, etc) must be determined. This compatibility, which we refer to as *"Vertical Compatibility"* is present if the transactional protocols (commit, recovery, global concurrency control) in the selected TS matches the corresponding protocols in the underlying resource managers (RM). Information for use when determining vertical compatibility can be gathered from the InfoBase component.

If vertical compatibility is present, the TS can be activated (if not already active). If the TS is not active, its compatibility with other active TSs must be determined. We refer to this as *"Horizontal Compatibility"*. Horizontal compatibility is only an issue if there are intersecting datasets between the current transaction and active transactions; i.e. transactions are working towards the same dataset. If there are no intersecting datasets, TS can be activated. Determining horizontal compatibility is related to the transactional guarantees provided by the active TSs. A *Synchronize TS Activation* procedure within TSActivate will determine intersecting datasets, check for horizontal compatibility, and control the TS activation. Executing a transaction is the last action in the Start_Trans() operation. The transaction is handed over to the selected TS deployed within the *TS Framework* where it will be executed by invoking an operation at one of TS Framework's provided interfaces.

TSActivate will implement the following algorithm when 'Start Transaction' is issued:

Start_Transaction($TransInfo_i$, Req_i):

```
    $TS_i$ = Selection_Procedure($Req$)
    If Check_Vertical_Compatibility($TransInfo_i, TS_i$) = OK Then
      Synchronize_TS_Activation($TransInfo_i, TS_i$)
      Start_Transaction($TransInfo_i, TS_i$)
    Else
      Return <Error Message>
 End
```

Synch_TS_Activation($TransInfo_i$, TS_i):

```
 If $TS_i \in$ List_Of_Active_TSs Then
   Return OK
 While $TS_i \notin$ List_Of_Active_TSs Then
   Intersection_DataSet = Current_DataSet $\bigcap$ Active_DataSets
   If Intersection_DataSet = $\emptyset$ Then
      <Insert TS into List_Of_Active_TSs>
 Else
      If Check_Horiz_Compatibility($TS_i$, $Intersection\_DataSet$) = OK Then
        <Insert TS into List_Of_Active_TSs>
    End
 End
```

This version a Synch_TS_Activation tries to activate a TS for an infinite period of time. To guarantee that a TS will be activated within an acceptable time period, other methods that can be replaced to hinder TS starvation and deadlock are discussed in [29].

The TSActivate component and its bindings (connections) to other components is illustrated in figure 7. An arrow from TSActivate to an interface provided by InfoBase means the TSActivate will issue requests on this interface.

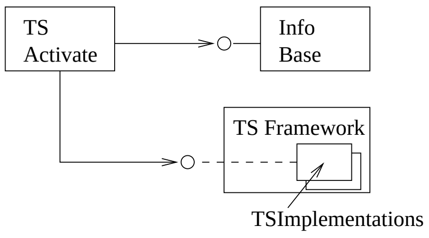

Fig. 7. The TSActivate component and its bindings

4.6 Configuration and Reconfiguration

Requests for configurations and reconfigurations of TS implementations are handled by the TSInstall component before they eventually are performed by the TS Framework where the TS implementations are deployed.

Modification of a transaction service can either involve a total replacement or a modification of one of its constituents. For ReflecTS, we suggest that modifying a part of a TS involves a replacement of the involved component(s). By this means, a reconfiguration of a TS simply involves removing and inserting service components.

The procedures undertaken by the TSInstall component will then be to deploy and remove service components (SC) and transaction services (TS), and to assemble TSs with deployed SCs. The TSInstall component with its bindings (connections) is depicted in figure 8.

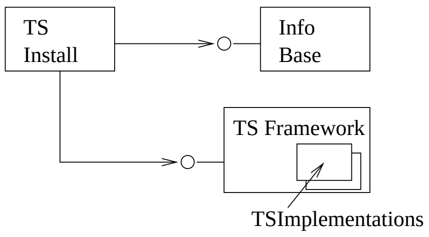

Fig. 8. The TSInstall component and its bindings

DeployTS() Firstly, TSInstall issues a request to the meta-interface provided by the TS Framework in order to deploy the TS. TS Framework assures that the TS conform to a valid architecture. Secondly, TSInstall invokes an interface provided by the InfoBase component to store the transaction service descriptor and compatibility information.

```
Deploy_TS($TS_i$, $TSDescr_i$):
 If TS_Framework.Deploy($TS_i$) = OK
    InfoBase.Store_TS_Descr($TSDescr_i$)
 Else Return($Msg : InvalidTS_i$)
```

RemoveTS(). Removing a TS can only be done if it is not in use by any transaction. Several alternative situations must be considered when implementing this task. *1)* The TS is in use and a queue of transactions are pending for it. If another TS is to be deployed as a substitute, the queue of waiting transactions can be transferred to the new TS or they can be finished by the old one. The new TS will be deployed and activated, and the old one removed. *2)* The TS is in use. No new TS will substitute the old one. The TS can only be removed when pending transactions are completed. *3)* The TS is not in use by any transaction so it can be removed.

The evaluation of a new TS as a substitute for an old one, can for instance be performed by a transaction service designer. This paper will not investigate that particular task. Regarding the task of waiting until a TS is inactive before deleting it, can be more refined and sophisticated than the one presented in the following. The following algorithm implements RemoveTS().

```
Remove_TS($TS_i$, $T$):
 While $TS_i \in$ ActiveTS Wait($Period$)
 TS_Framework.Remove_TS($TS_i$)
 InfoBase.Delete_TS_Descr($TSDescr_i$)
```

DeploySC() and ***RemoveSC()*** are operations used in accordance with TS configuration or reconfiguration. As we have suggested, a TS reconfiguration involves the replacement of one or more of its constituents. Consider for instance replacing a commit protocol. When the TS is inactive, the old service component is removed using RemoveSC() and the InfoBase updated. A new commit protocol is inserted issuing InsertSC(). The TS Framework assures that the reconfigured TS conforms to a valid configuration. Then the InfoBase is updated with information about the newly inserted SC and the reconfigured TS.

AssembleTS() is an operation for composing TSs using deployed service components. Information about the service components, stored in the InfoBase, are used in the composition process. After composing a new TS, the InfoBase component is updated.

5 Implementation; Architecture and Issues

This section presents an architecture for a prototype implementation of ReflecTS which we are currently working on. The implementation uses OpenCOM components [13] and the ReMMoC component framework model [19] as its building blocks. The OpenCOM component model is build upon COM with added support for reflection via provided meta-interfaces. This makes OpenCOM an ideal building block for reflective middleware, which so far has been shown in the OpenORB [9] and the ReMMoC [21] implementations. Both OpenORB and ReMMoC describes component framework models which are suitable for configurable and reconfigurable component systems. The ReMMoC component framework is however the most generic one, offering a complete meta-interface for architectural reflection (inspection and dynamic adaptation) and a domain-specific method for constraining the scope of evolution. The meta-interface provided by a ReMMoC CF, the ICFMetaArchitecture interface, see appendix A, has shown sufficiently for configuration and reconfiguration of TS implementations as proposed in this paper. We have also shown the possibility of switching between TS implementations.

ReMMoC CF provides integrity maintenance in the face of dynamic changes. After configuring or reconfiguring a CF it must be checked to ensure that it provides the correct functionality. To to this, each CF provides a connection to a *Accept* component into which developers can plug in their own checking implementation. We have explored this technology to specify architectures for valid TS implementations. This has so far been successful, and changes to the architecture which are not valid are rejected. Integrity maintenance also assures that changes are performed at appropriate times. Therefore, each component framework provides a readers / writers lock to access the local CF graph. Any call to change the configuration of the CF, accesses the lock as a writer.

Figure 9 gives an overview of the ReflecTS platform.

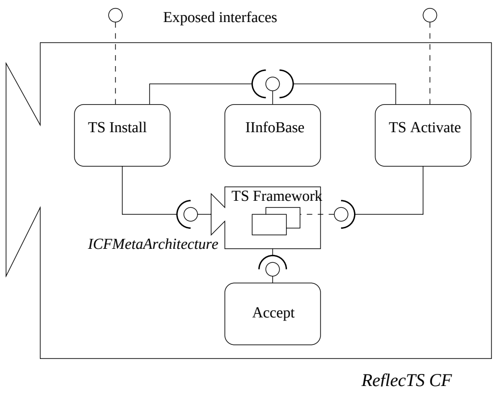

Fig. 9. Overview the ReflecTS platform

5.1 Implementation Issues

An implementation of ReflecTS must take into consideration a lot of problems and decisions raised by the applications, the middleware layer and the underlying system. We will in the following present some of them shortly.

Interface between Applications and Transactional Middleware Traditionally, applications conforms to the X / Open standard and uses the TX-interface [25] which is a standard interface between applications and GTM / transactional middleware. Some investigation has to be done in order to uncover whether the TX-interface is adequate enough within an extendible and flexible transactional middleware model or whether a new interface should be described.

Specifying Global Transactions Some information must be provided along with the issue of a transaction. Work has to be done in order to decide what kind of information and how to describe it. The information can for instance include a specification of the transaction structure, the transactional requirements and directions for which resource managers to use.

Specifying Transactional Requirements Applications needs a programming model for specifying transactional requirements. Transactional properties ACID and other properties must be gradually refined into more specialized properties. [47] refines the ACID properties using temporal logic, and the refined properties then serve to characterize the behavior of various transaction models. [16] introduces the definition of *atomicity spheres* to capture a variety of notions of atomicity. Specifying requirements can also be done by using a xml-file, a specific language, or temporal logic as proposed in [47][48].

Composing Transaction Services Within the ReflecTS platform we suggest composing TS's using a number of components. The first question that arise is *how* to split a TS into adequate components. Thereafter, composing a TS presupposes a set of rules constraining the composition to assure a valid TS. For instance, when composing a TS using commit, recovery and global concurrency control, those protocols must be matching in order for a TS to be valid.

Describing Transaction Services Deployed transaction services must be described in order to distinguish between them. This must be done in accordance to the way transactional properties are described so that the selection process can perform the merging process. Describing transaction services can be done using for instance ACTA [12, 11] or XML.

Transaction Service Selection Given the description of transaction services and a formal specification of transactional properties, a selection procedure selects a TS for a given transaction. The procedure accepts a set of transactional requirements and verifies whether they are supported by existing transaction services, TSs. This procedure depends on formal descriptions of transactional requirements and transaction services.

TS Selection; Vertical Compatibility A TS implements transactional mechanisms (commit, recovery and global concurrency control (GCC)), which must be vertically compatible with corresponding mechanisms in present underlying RMs. Protocols used for concurrency control, commit and recovery in underlying RMs must be "localized" in one way or another so that "matching" protocols can be implemented in a TS. GCC and commit / recovery for global transactions in heterogeneous multi-database systems [45][10] are problematic issues related to both heterogeneity and autonomy. Combining different concurrency and commit / recovery schemes must be done carefully. [4] addresses the issue of compatibility among atomic commit protocols when different sites in a distributed environment uses different protocols. Assuring global serializability when local DBMSs uses different concurrency control protocols is also problematic, as global serializability can only be guaranteed when the different sites runs the same protocol.

TS Activation; Horizontal Compatibility Activating a TS depends on its compatibility with other active TSs. We refer to this as horizontal compatibility as we are talking about compatibility between services at the same level. Determining horizontal compatibility is only necessary if concurrent transactions are working towards the same dataset. Horizontal compatibility between TSs depends on their provided transactional guarantees, which TS was activated first, and the applications perception of consistency (i.e. what is consistent for one application may not be consistent for another one) [29].

TS Configuration and Reconfiguration Configuration and reconfiguration of a TS must conform to a valid configuration. Validation can be performed by either checking the TS towards a set of valid configurations or by manually interacting with a transaction service designer. Reconfiguration of transaction services must also be done at appropriate times, which preferable means when the TS is inactive. Otherwise, the results could cause inconsistencies.

Interface between TS and RMs Traditionally, the X/Open XA-specification [24] defines the standard interface between transaction manager and RM's. X/Open XA provides a *prepare-to-commit* event to simplify the execution of distributed commit. The X/Open XA-interface contains methods that can capture a broad range of transaction management even though the interface is statically defined. For instance, a variety of different commit-protocols performs satisfactorily by using the XA-interface. Some work has to be done in order to uncover whether the XA-interface is sufficient for a transaction processing environment characterized by flexibility and unpredictability.

Advertising Transaction Services Advertising changes within the pool of transaction services is a task not affected by this work. Knowledge of new, updated, or deleted transaction services can be obtained manually by the application or the application designer either on a need-to-know basis or feeded in. Alternatively this can be done with the help of a publish-subscribe protocol.

6 Related Work

While the concept of reflection was originally introduced in the area of programming language design [41], it is now a concept used in different areas to "open up" systems. Reflection is for instance used in operating systems [1], distributed systems [42], middleware [8], and now also in transactional systems [6, 46].

Emerging reflective middleware platform utilize the concepts of open implementation and reflection to get access to the underlying virtual machine. A number of such platforms have been developed, including OpenORB [9], Dynamic TAO [40], Open CORBA [33], and Flexinet [26]. These platforms offer great flexibility in terms of meeting the needs of application domains. However, they do not provide transactional services.

In [6] a Reflective Transaction Framework is described, that implements extended transaction models on top of TP-monitors. The framework uses transaction adapters on the meta level to extend TP-monitor behaviour. The adapters, which include the transaction management logic for an extended model, are given the control over transaction processing at certain transactional events.

[46] describes how Reflective Java can be used to implement a flexible transaction service. It allows application developers to provide application-specific information to a container so that this can be used to customize the transaction service. The framework enables a container to change its functionality by plugging/unplugging its metaobjects, and thus be customized to meet new application requirements or changing environment conditions.

Besides the related work on reflective transaction services, our work also relates to research on dynamic combination and configuration of transactional and middleware systems. The work of [47, 38, 37, 11] recognizes the diversity of systems and their different transactional requirements, and describes approaches to how these diverse needs can be supported.

In [47] a formal method to synthesis transactional middleware is specified. The work describes an approach that takes transactional requirements for a given system as input, selects available service components and composes a transactional middleware customized to the needs of the system. [38] argue the necessity to allow both design time and runtime specification of transaction models. Transaction model elements are organized such that parts of the specification can be done before transactions are executed, while the remaining parts can be specified during runtime. Runtime specification of transaction executions are done by users. [37] proposes an extension of the transaction concepts in EJB, called Bourgogne transactions, that adds a set of advanced transactional properties allowing some flexibility in transaction executions. In [11] the ACTA framework is used as a tool to support the development and analysis of new extended transaction models. However, implementing a model specified in ACTA is up to the developer.

Our work on a reflective transactional system contrasts previous work on two matters: Firstly, we focus on how to guarantee correctness for the reflective transactional system. The close relationship between different transaction service modules (for instance commit-, recovery- and global concurrency control), makes it necessary to control correctness of the transaction service if the internals of the service is manipulated. Secondly, we allow the use of a number of concurrently active transaction services, and must guarantee consistent use of possibly incompatible services.

7 Conclusion

In this paper we have argued that transactional middleware must meet an increasingly diverse range of requirements from open applications. To accommodate these requirements, we have designed *ReflecTS*, a Reflective Transaction Service platform, providing configurability, reconfigurability and concurrently running transaction services. To our knowledge, this problem has not been addressed by previous works.

ReflecTS is designed using reflection, components and component frameworks as the underpinning technologies. The ReflecTS platform provides: concurrently running transaction services, policies for assuring correctness for concurrently running transaction services; meta-interfaces for inspection and adaptations of the internals of the platform; policies for constraining the scope of configurability and reconfigurability of transaction services.

We are currently working on a prototype implementation of ReflecTS, using OpenCOM components [13] and the ReMMoC component framework model [19] as presented in section 5. At the present time the synchronization protocol and the compatibility issues are further explored. A programming model to classify and specify transactional requirements will also be included. Our work continues and will be extended to include solutions to some of the issues listed in section 5.

Appendix A

A ReMMoC CF implements the ICFMetaArchitecture interface to achieve access to internal structure (reflection) for configuration and reconfiguration. The table below lists operations provided by the ICFMetaArchitecture interface.

Operations for Inspection	*Description*
get_internal_components	Returns a list of the identifiers of the components that constitute the base-level configuration
get_bound_components	Returns a list with information of all components bound to the one identified as the argument
get_internal_bindings	Returns a list with ids of all binding objects that are part of the base-level composition

Operations for Reconfiguration	*Description*
local_bind	Establish a local binding between the two identified interfaces
break_local_bind	Break the local binding between the two interfaces
insert_component	Create and insert a component into the base-level configuration with the given name and in the specified location (given by zero or more interfaces to which the new component should be bound, if zero interfaces, the new component is left unbound)
remove_component	Delete the component from the configuration, re-binding the adjacent interfaces of neighboring components, if appropriate and according to the given mapping of interfaces to be rebound
replace_component	Replace an existing component with a new component of the given type (deleting the old component)
Expose_Interface	Map the interface of an internal component as a new interface of the composite CF
UnExpose_Interface	Remove an exposed interface
Expose_Receptacle	Map the receptacle of an internal component as a new receptacle of the composite component
UnExpose_Receptacle	Remove an exposed Receptacle
ReplaceConfiguration	Replace the current graph of components with a new graph
init_arch_ransaction	Start the transaction for architecture reconfiguration
commit_arch_transaction	Completes the reconfiguration
rollback_arch_transaction	Rolls back any changes made during an architectural transaction

References

1. *The Apertos Reflective Operating System: The Concept and Its Implementation*, October 1992.
2. Corba services, transaction service specification, v1.1, 1997.
3. Robert K. Abbott and Hector Garcia-Molina. Scheduling real-time transactions: a performance evaluation. *ACM Trans. Database Syst.*, 17(3):513–560, 1992.
4. Yousef J. Al-Houmaily and Panos K. Chrysanthis. Atomicity with incompatible presumptions. pages 306–315, 1999.

5. A. Andersen, G. Blair, V. Goebel, R. Karlsen, T. Stabell-Kulø, and W. Yu and. Arctic beans, configurable and re-configurable enterprise component architectures. In *IFIP/ACM International Conference on Distributed Systems Platforms*, Heidelberg, Germany, 2001. Middleware.
6. R. Barga and C. Pu. Reflection on a legacy transaction processing monitor, 1996.
7. Gordon S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, London, 1998. Springer-Verlag.
8. Gordon S. Blair and Geoff Coulson. The case for reflective middleware. Distributed Multimedia Research Group Report MPG-98-38, Distributed Multimedia Research Group, Lancaster University, 1998.
9. Gordon S. Blair, Geoff Coulson, Anders Andersen, and Lynne Blair et.al. The design and implementation of open orb 2. *DSOnline*, 2(6), 2001.
10. Yuri Breitbart, Hector Garcia-Molina, and Abraham Silberschatz. Overview of multidatabase transaction management. *VLDB Journal: Very Large Data Bases*, 1(2):181–293, 1992.
11. Panos K. Chrysanthis and Krithi Ramamritham. Synthesis of extended transaction models using acta. *ACM Transactions on Database Systems*, 19(3):450–491, 1994.
12. Panos K. Chrysanthis and Krithi Ramaritham. Acta: A framework for specifying and reasoning about transaction structure and behavior. In *Proccedings of the 1990 ACM SIGMOD international conference on Management of data*, May 1990.
13. Michale Clarke, Gordon S. Blair, Geoff Coulson, and Nikos Parlavantzas. An efficient component model for the construction of adaptive middleware. In *Middleware*, Heidelberg, Germany, 2001.
14. Microsoft Corporation. The component object model specification, version 0.9. Technical report, October 1995.
15. Microsoft Corporation. The .net framework, 2000.
16. W. Derks, J. Dehnert, P. Grefen, and W. Jonker. Customized atomicity specification for transactional workflows. Tehnical Report TR-CTIT-00-24, University of Twente, 2000.
17. Margaret H. Dunham, A. Helal, and S. Balakrishnan. A mobile transaction that captures both the data and movement behaviour. *ACM-Baltzer Journal on Mobile Networks and Applications (MONET)*, 2(2), 1997. Kangaroo transactions.
18. Ahmed K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, 1992.
19. Paul Grace. *Overcoming Middleware Heterogeneity in Mobile Computing Applications*. PhD thesis, Lancaster University, 2004.
20. Paul Grace, Gordon S. Blair, and Sam Samuel. Interoperating with services in a mobile environment. Technical Report MPG-03-01, Lancaster University, 2003.
21. Paul Grace, Gordon S. Blair, and Sam Samuel. A marriage of web services and reflective middleware to solve the problem of mobile client interoperability. In *In Proceedings of Workshop on Middleware Interoperability of Enterprise Applications (MIEA)*, Dublin, Ireland, September 2003.
22. Object Management Group. The common object request broker: Architecture and specification. Technical Report Tech. Report 96.3.4 (Revision 2.0), Object Management Group, July 1995.
23. Object Management Group. *CORBA Component Model, V30*, 2004.
24. The Open Group. The x/open cae specification. distributed transaction processing: The xa specification. x/open document number: Xo/ca/91/300, December 1991.
25. The Open Group. X/open cae distributed transaction processing: The tx specification, December 1995.
26. R. Hayton. Flexinet open orb framework. Acm technical report 2047.01.00, APM Ltd, Poseidon House, Castle Park, Cambridge, UK, 1997.
27. Apple Computer Inc. Opendoc: White paper. Technical report, Appel Computer Inc., 1994.
28. Oberon Microsystems Inc. Blackbox developer and blackbox component framework. Technical report, Oberon Microsystems Inc., 1997.
29. Randi Karlsen. An adaptive transactional system - framework and service synchronization,. In *International Symposium on Distributed Objects and Applications (DOA)*, Catania, Sicily, November 2003.
30. Randi Karlsen and A. B. A. Jakobsen. Transaction service management an approach towards a reflective transaction service. In *2nd International Workshop on Reflective and Adaptive Middleware*, Rio de Janeiro, Brazil, June 2003.
31. Gregor Kiczales, J. des Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
32. Gregor Kiczales, John Lamping, Cristina Videira Lopes, chris Maeda, Anurag Menedhekar, and Gail Murphy. Open implementation design guidelines. *ACM*, 1997.
33. T. Ledoux. Implementing proxy objects in a reflective orb, 1997.

34. Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of the Conference of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, oct 1987.
35. Sun Microsystems. The java beans specification 1.01, 1997.
36. Sun Microsystems. Enterprise javabeans specification version 2.0, final release, 2001.
37. Marek Prochazka. Advanced transactions in Enterprise Java Beans. *Lecture Notes in Computer Science*, 1999:215–??, 2001.
38. Heri Ramampiaro and M. Nygaard. Cagistrans: Providing adaptable transactional support for cooperative work. In *Proceedings of the 6th INFORMS conference on Information Systems and Technology (CIST2001)*, 2001.
39. K. Ramamritham and P.K. Chrysanthis. *Executive briefing: Advances in concurrency control and transaction processing*. IEEE Computer Society Press, Los Alamitos, California, 1997.
40. M. Roman, F. Kon, and R. Campbell. Design and implementation of runtime reflection in communication middleware: the dynamictao case, 1999.
41. B.C. Smith. *Procedural Reflection in Programming Languages*. PhD thesis, MIT, MIT Computer Science Technical Report 272, Cambridge, 1982.
42. Robert Stroud. Transparency and reflection in distributed systems. *ACM SIGOPS Operating Systems Review*, 27, 1993.
43. Allarmaraju Subhramanyam. Java transaction service, 1999.
44. Clemens Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1997.
45. Gerhard Weikum and Hans-Jorg Schek. Concepts and applications of multilevel transactions and open nested transactions. In *Database Transaction Models for Advanced Applications*, pages 515–553. 1992.
46. Zhixue Wu. Reflective java and a reflective component-based transaction architecture. In *OOPSLA workshop*, 1998.
47. A. Zarras and V. Issarny. A framework for systematic synthesis of transactional middleware, 1998.
48. A. Zarras and V. Issarny. Imposing transactional properties on distributed software architectures, 1998.