Computer Technology and Application 7 (2016) 173-195
doi: 10.17265/1934-7332/2016.04.001

TransJ: An Abstract Independent-Framework for Weaving Crosscutting Concern into Distributed Transactions

Anas M. R. AlSobeh[1] and Stephen W. Clyde[2]

1. Department of Computer Information Systems, Yarmouk University, Irbid, Jordan

2. Department of Computer Science, Utah State University, Logan, UT 84322, USA

Abstract: Implementing crosscutting concerns for transactions is difficult, even using aspect-oriented programming languages such as AspectJ. Many of these challenges arise because the context of a transaction-related crosscutting concern consists of loosely-coupled abstractions like dynamically-generated identifiers, timestamps, and tentative value sets of distributed resources. Current aspect-oriented programming languages do not provide joinpoints and pointcuts for weaving advice into high-level abstractions or contexts, like transaction contexts. To address these problems, we propose an extension to AspectJ framework, called *TransJ*, that allows developers to define pointcuts in terms of transaction abstractions and that automatically keeps track of context information for transactions. This paper describes *TransJ* as an abstract independent framework for weaving crosscutting concerns into high-level runtime abstractions, with which developers can implement transaction-related crosscutting concerns in modular, cohesive and loosely coupled transaction-aware aspects. Finally, this paper presents eight different ways in which *TransJ* can improve the reuse with preserving the performance of applications requiring transactions. Informally, these hypotheses are that *TransJ* yields (1) better encapsulation and separation of concern; (2) looser coupling and less scattering; (3) higher cohesion and less tangling; (4) reduces complexity; (5) improves obliviousness; (6) preserves efficiency; (7) improves extensibility; and (8) hastens the productivity. A brief discussion of experiment to test the hypotheses is provided, but the details of the experiment are left for another paper.

Key words: AOP, AspectJ, transaction, joinpoint model, context, crosscutting concern, encapsulation, high-level abstractions, modularity, reuse.

1. Introduction

DTPSs (Distributed transaction processing systems) can be unnecessarily complex when crosscutting concerns, e.g., logging, concurrency controls, transaction management, and access controls, are scattered throughout the transaction processing logic or tangled into otherwise cohesive modules. A challenge with the implementation of DTPS is that some properties and functionalities cannot be easily encapsulated and localized into loosely coupled abstractions, which increases the complexity of the system. Other challenges stem from the essential complexity in the nature of the data, operations on the data, or the volume of data, and accidental complexity comes from the way that the problem is being solved, even using common transaction frameworks [1].

OO (Object Orientation) encourages encapsulation of design decisions and therefore leans towards distributing responsibilities across the various types of objects. OO has proven to be effective in modeling common hierarchical behaviors, but falls short in modeling behaviors that span (i.e., crosscut) multiple unrelated modules (i.e., contexts) [2]. Attempts to implement such crosscutting concerns in OOP (Object-Oriented Programming) often result in systems that are difficult to reuse or maintain: this is where AOP (Aspect-Oriented Programming) comes in

Corresponding author: Anas AlSobeh, Ph.D., assistant professor, research fields: aspect-oriented programming and distributed systems.

Ref. [3]. AOP encapsulates crosscutting concerns in first-class software components, called *aspects* [3]. An aspect is an ADT (Abstract Data Type) and very much like a class in OOP and an aspect instance is like an object, except that an aspect defines special methods, called *advices*, which are automatically woven into the core application according to specifications, called *pointcuts*. A pointcut identifies a set of *joinpoints*—a logical intervals in the execution flow of the system where and when weaving of advice takes place. Each joinpoint begins and ends relative to static places in the source code, called *shadows* [4]. Weaving is the process of composing core functionality modules with aspects, thereby yielding a working system [3]. However, the difference between AOP and OOP is that AOP offers better abstractions for separating crosscutting concerns from core functionality that do require core functionality to dependent on crosscutting concerns in any way. An AO (Aspect-Oriented) developer should be able to add/remove aspects to/from a project without changes to any other code. Some authors refer to this as a principle, called *obliviousness* [6].

AspectJ is an extension to the JPL (Java Programming Language), which provides separate mechanisms for defining an aspect and specifying its interaction with an underlying system [7]. It allows application programmers to weave advice for the logic of crosscutting concerns into the execution of code-based contexts, such as constructor calls/executions, method calls/executions, class attribute references, and exceptions [8]. AspectJ, like many other existing AOPLs (AOP Languages) and frameworks, suffers from the lack of capabilities that would handle high-level runtime abstractions; therefore, it does not directly allow behaviors to be woven into more abstract contexts, such as transactions. The transaction represents a major crosscutting concern in DTPSs because it is difficult to encapsulate and modularize with current technologies. Even though transactions core concepts in many distributed systems, they are rarely treated as a first-class programming concept. Consequentially, the logic for transactions is, in general, scattered or spread across several units of the DTPS [12]. Thus, when changes occur to that logic, there can be a large ripple effect on the whole system.

A transaction is a set of operations on shared resources, such that its execution results in either the successful completion of all operations or the completion of no operation. Besides this all-or-nothing property, called *atomicity*, transactions are *consistent*, *isolated*, and *durable*, meaning that persistent data will only change from one valid state to another; that other concurrent transactions cannot see the effects of a transaction until it is completed; and, that effects of a transaction become persistent after completion, even if there is system failure. Together, Atomicity, Consistency, Isolation, and Durability are often referred to as the ACID properties [6].

Distributed transactions are transactions, but their operations execute on multiple host machines, ideally with improved throughput. From a logical perspective, a distributed transaction can be a flat sequence of operations or a hierarchy of sub-transactions, also known as nested transactions. In the latter case, nested transactions may execute concurrently and sequentially.

Regardless of whether a distributed transaction is a flat or nested transactions, it is an ephemeral concept that spans multiple execution threads and operations and may use a variety of distributed resources. Therefore, from an execution-timeline perspective, it may seem non-contiguous and unevenly spread out. A transaction's context is not tied to code constructs, like constructors and methods, in a single thread of execution; rather, it consists of loosely coupled abstractions like dynamically generated identifiers, timestamps, and tentative value sets for distributed resources. This makes it very difficult for aspect-oriented developers to localize and encapsulate crosscutting concerns that apply to transactions as

execution units.

This paper provides a foundation for developing an extension to AspectJ, called *TransJ*, that allows application programmers to weave aspect behaviour for transaction-related crosscutting concerns into a DTPS in a productive, modular and reusable way, while preserving performance, core functionality and obliviousness to crosscutting concerns. *TransJ* offers a framework, independent the underlying transaction framework that allows Aspect-Oriented developers to treat transactions as first-class concepts into which compilers can weave transaction-related crosscutting concerns. Specifically, it defines interesting time points/places for when/where the crosscutting concerns might augment an application's core functional or the underlying transaction processing system. To establish this extension, we captured key transaction events and context information in a conceptual model, called UMJDT (Unified Model for Joinpoints of Distributed Transactions) [15]. The implementation perspective of *TransJ* utilizes the JTA (Java Transaction API) standards, which is the de facto standard in the Sun *Java Enterprise Edition* (J2EE) for handling distributed transaction development today. Section 2 provides a high-level overview of a conceptual model that provides a theoretical foundation for *TransJ*, namely its transaction jointpoints and contexts. Section 3 provides a high-level explanation of *TransJ* architecture. Section 4 discusses the lower-level design, implementation of *TransJ*, base aspects central to core *TransJ*'s implementation. To validate *TransJ*, we have created a library of reusable aspects for common transaction-related crosscutting concerns and have applied them to a variety of sample systems and then it discusses how application programmers can write their own transaction aspects. Based on initial theoretic notions, we hypothesize that developers should see reuse improvements while preserving the software performance relative to eight hypotheses discussed in Section 6. Related work is presented in Section 7. Finally, Section 8 summarizes the current state of *TransJ* and outlines our future work.

2. A High-Level Overview of Umjdt

Fig. 1 shows part of the UML model, called the UMJDT (Unified Model for Joinpoints in Distributed Transactions). It describes a common conceptual understanding about transactions to encapsulate any complex relationship, which can exist in a DTPS, and

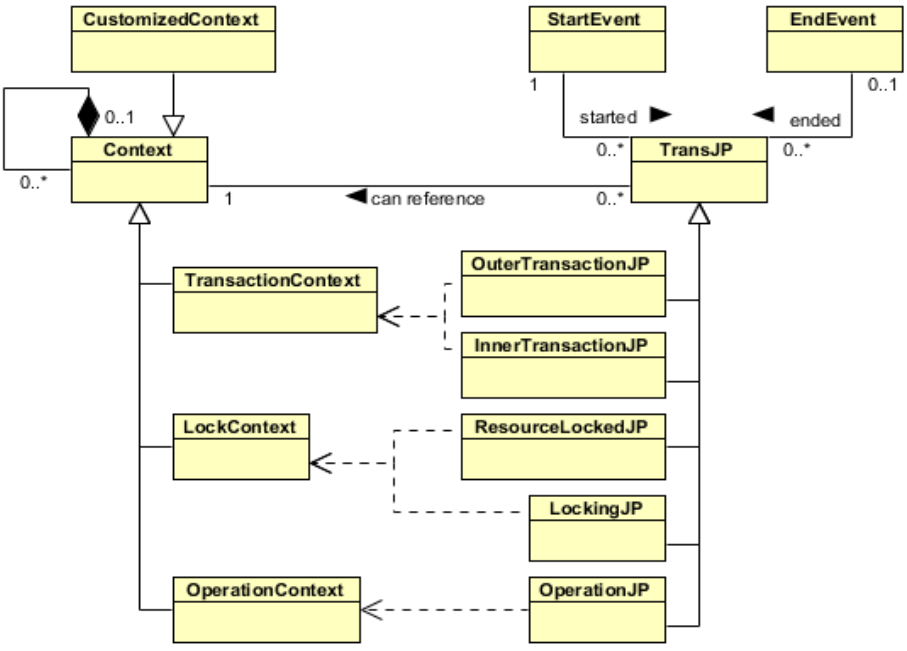

Fig. 1 Part of the Unified Model for joinpoints in distributed transactions.

captures the key ideas for new transaction joinpoints and related context information. Specifically, it unifies DTPS concepts related to (a) transactions in general, (b) the kinds of information that comprise their context, and (c) events that represent interesting time points/places for when/where the crosscutting concerns might augment an application's core functional or the underlying transaction processing system.

Overall UMJDT describes transaction-related joinpoints and context information that make the most sense for DTPS's, therefore, it's considered as a foundation for weaving transaction aspects into high-level abstractions, i.e., transactions, in *TransJ*. The implementation of *TransJ* included an implementation of UMJDT that provided the ability to weave advice into transaction program execution before, after, or around complete transactions or individual transaction operations.

In general, UMJDT serves as a base for formalizing transaction joinpoints, which fall into three general categories: transaction joinpoints, operation joinpoints, and concurrency control joinpoints. They are represented five new types of joinpoints for DTPS's: outer transaction, inner transaction, resource locked, locking, and operation joinpoints. These joinpoints represent alogical intervals of time in a flow of execution and have a beginning and an end. Each type of joinpoint is referenced to a specific context that holds all the relevant statics and runtime information for it. For more details on the UMJDT design and examples are given in Ref. [15].

3. A High-Level Overview of *TransJ*'s Architecture

TransJ represents a set of principles that provide an independent abstract framework, which enables separating of complex transaction concerns into manageable, cohesive and coherent concepts. Fig. 2 shows an architectural block diagram of *TransJ*, in which the colored blocks represent relevant conceptual layers, and arrows depict dependencies among these layers. It describes the *TransJ*'s design at a higher level, with adopting a strategy of top-down to the design of the *TransJ* with a layered architectural design [16], in which each layer embodies a reusable function or the logical component and provides services to the layer above it and uses the services of the layer below it. Thus, *TransJ* enables aspect-oriented developers to treat transactions as first-class concepts into which AspectJ framework can weave crosscutting concerns in a modular way, i.e., transaction aspects. This promotes greater enhancements, obliviousness, localization along with

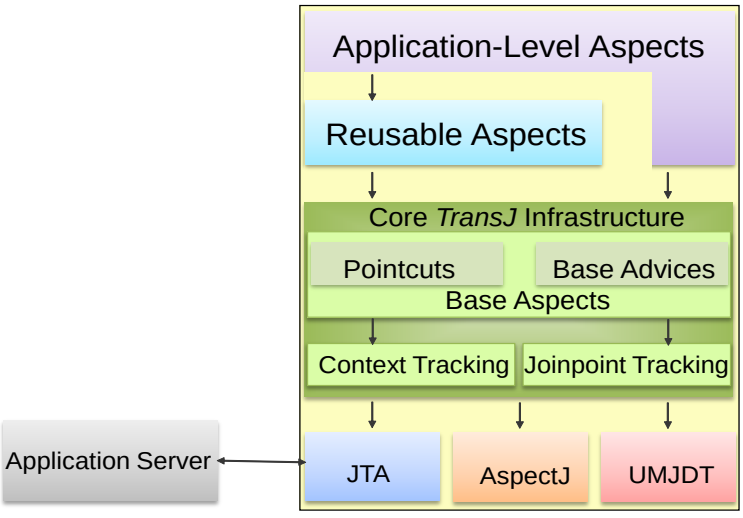

Fig. 2 The architectural pattern to *TransJ*.

code reusability while preserving the performance. The following sub-sections provide some necessary details about each of the layers in the order from top to bottom, that will ultimately set the stage to assess whether the benefits are achieved. The following sub-sections provide some necessary details about each of the layers.

3.1 Application-Level Aspect Layer

This layer is an abstraction layer that contains a set of common transaction-related aspects, which encapsulate base-application requirements. The aspects of this layer are aspects of aspects. In other words, we can build application-level aspects either by extending the abstract aspects provided by the reusable aspects or/and base aspects in core *TransJ*.

Application-level aspects can use directly either the base aspects or the abstractions provided by *TransJ* to access metadata that related to transactions, operations and the affected resources that are pulled by context trackers. This type of information can describe the expected behavior of the transaction with respect to the context in which it is running such as lock context, operation context, or transaction context [15].

In [@@reference@@], we show that application-level aspects that are easy-to-code, more reusable, understandable, predictable, flexible and modular than similar concerns, which programmed in AspectJ or OOP approach.

3.2 Reusable Aspect Layer

The definition of the reusable layer within the scope of *TransJ* is a layer responsible for providing a set of helpful aspects that encapsulate common transaction-related crosscutting concerns and exposing relevant context data that application aspects must consider once weaving advices. Overall the reusable aspects represent general crosscutting concerns commonly found in applications with significant transaction requirements, and therefore can be woven in DTPSs where a transaction-related concern is applicable. In other words, this layer represents a toolkit-like collection of transaction aspects that developers should find useful for in several of DTAs (Distributed Transaction Applications). These reusable aspects depend on a set of the core *TransJ* aspects that can decrease the development time to program application-level aspects, and make them more understandable, reusable, predictable, and oblivious. To ensure that is done effectively, we need appropriate, precise specifications of such aspects that can then be used to understand the behavior of the DTPSs and introduce behaviors into complex like nested distributed transactions. The core *TransJ* provides specifications for reusable aspects. This kind of aspect is conceptually inspired from the key transaction joinpoints defined in the UMJDT.

3.3 Core TransJ Infrastructure Layer

The core *TransJ* is a library that introduces a transaction JPM on top of AspectJ JPM. It consists of components for tracking transaction contexts and joinpoints; base aspects that core transaction abstractions; and a collection of pointcuts for transaction events and operations. The base aspects include base advices that embody and augment the behavior of a transaction; and a collection of pointcuts for gathering context information that can be used in the advice code.

We specify the behavior of transaction aspects in terms abstraction concepts that to any DTPS built using JTA and XA specifications [18]. These abstract aspects define one or more pointcuts and items of advices that will execute when transaction reaches joinpoints matching these pointcuts.

The Context and Joinpoint tracking (i.e., trackers) encapsulate hooks into the underlying transactions subsystems, such as JTA transaction and UMJDT transaction, in which pull relevant context information for transaction base aspects and keep track the start and end points of the joinpoint. If those changes, one only needs to replace or extend these trackers. The

base aspects make use of the context information provided by the context tracking and allow reusable or application-level aspects specific to individual transactions. The joinpoints defined in the *TransJ* core infrastructure give the reusable aspect and application-level aspect convenience, reusable pointcuts for transactional joinpoints. A software developer that wants to use transaction-related aspects simply has to include this library in the project.

3.4 JTA (Java Transaction API)

JTA is another foundation part of *TransJ* architecture. It offers a procedural interface to transactions and resources, including several methods that allows an application programmer to start, join, commit, and abort transactions [18]. Begin operation, which starts a new transaction or a nested transaction within an already ongoing one; commit operation, which attempts to commit the current transaction; abort operation, which forces the transaction to rollback, and more. In addition, it provides multithreaded transaction models provide additional operations to allow threads to join an ongoing transaction (join operation), which allows the calling thread to join the transaction with the current transaction context. *TransJ's* pointcuts tied to the JTA constructs.

4. Design and Implementation of a *TransJ* Tool Set

Fig. 3 shows a UML paradigm that represents the general architecture of *TransJ* along with some fundamental transaction-related concepts and functions. Specifically shows low-level aspects—a set of small well defined aspects that providing a specific cohesive sub-functionality, high-level joinpoints, high-level contexts, and trackers. The following subsections provide the details.

4.1 Transaction Joinpoints and Contexts

The motivation for the Joinpoints in *TransJ* rests on offering places and times where/when advice can be executed [15]. In AspectJ, they correspond to constructors, methods, attributes, and exceptions. In *TransJ*, they correspond to abstractions that may span into interleaved multi-threaded, modules or distributed hosts. The UMJDT serves as a foundation for formalizing transaction joinpoints, which fall into three general categories: transaction joinpoints, operation joinpoints, and concurrency control joinpoints. These categories refer to three different contexts: transaction context, operation context, lock context, respectively [15].

Fig. 3 presents a general joinpoint that is labeled by *TransJP* that encompasses the logical connection between transaction-event joinpoints. It is carried out generic transaction joinpoints, such as creating transaction-event joinpoints and finding where a specific transaction is involved. Each event can be associated with many other events, with at most one thread. One transaction can have multiple threads, and a host can process multiple transactions concurrently. For example, in a distributed nested transaction system, a transaction *T1* can begin executing on the thread *Th#1* which corresponds to a begin event, and then allows the transaction to commit or abort for some other thread *Th#2*.

The green boxes in the figure are *TransJ* classes that implement joinpoints for different kinds of contexts. Such joinpoints offer a natural abstraction in term of events, enable the explicit definition of complex crosscuts by means of event pattern, and accommodate very general behaviors of a transaction.

Overall *TransJP* represents a joinpoint for the entire execution transaction, as well as joinpoints for a sequence of sub-transactions within a transaction scope, for a sequence of operations within an operation scope, and for lock/release concurrency operations within a lock scope. *TransJP* defines three event types: begin event, commit event, and abort event. The begin event is when something happens at a particular point, i.e., begin point, related to the setting

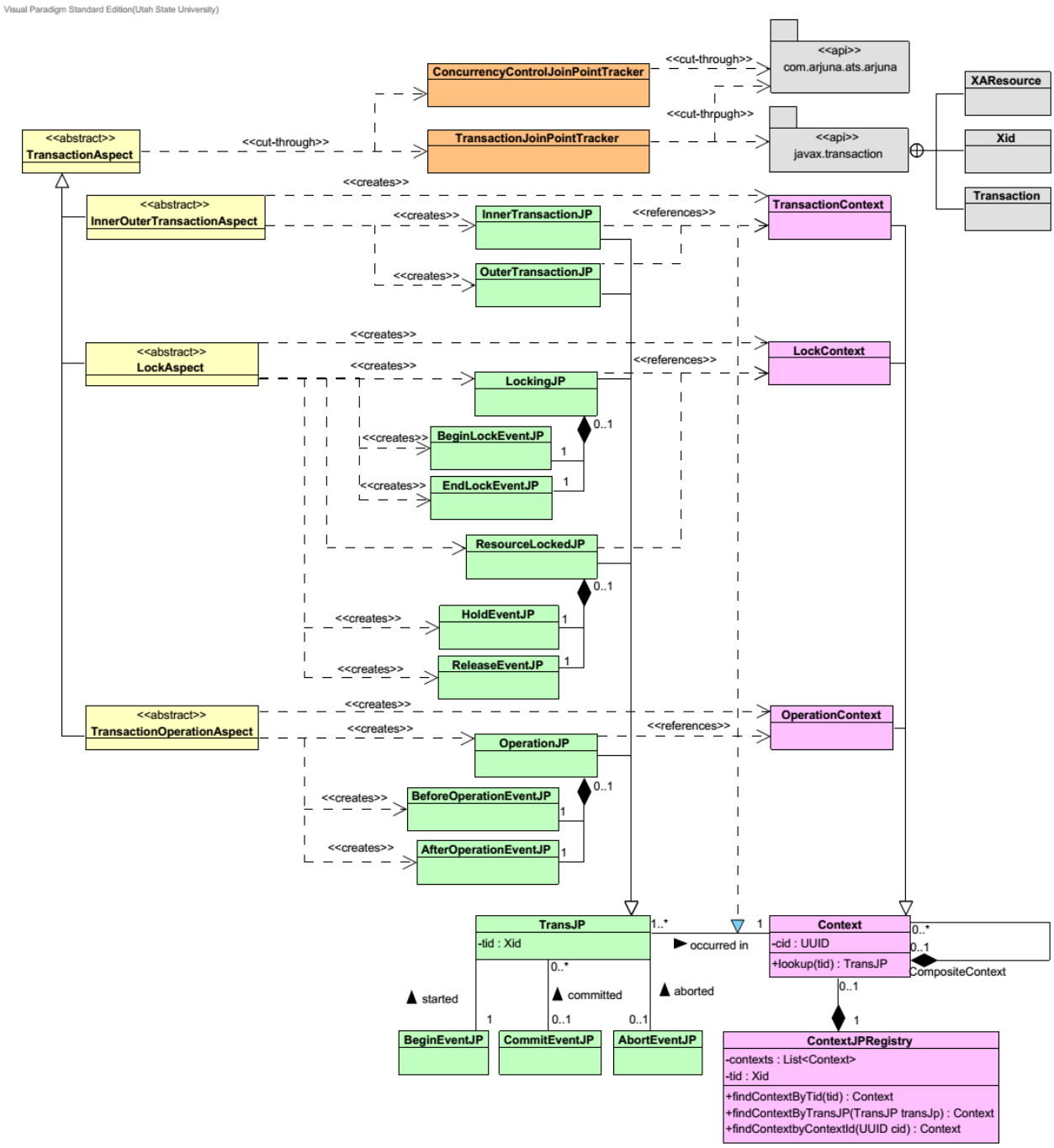

Fig. 3 Part of the *TransJ* Event, joinpoint, context and aspects.

up of transaction flow control. The commit or abort event is when something happens at a particular point, i.e., commit or abort point, related to the end of the transaction execution flow. These events are mapped to three event joinpoints, respectively: BeginEventJP, CommitEventJP or AbortEventJP. Each one implements a single joinpoint for an individual transaction event. BeginEventJP represents an execution point of the code into which advice can be woven, when *TransJP* related to the begin event of the transaction occurs in the transaction system. CommitEventJP or AbortEventJP represents an execution point of the code into which advice can be woven, when *TransJP* related to the commit or

rollback event transaction, respectively, occurs in the transaction system. *TransJP* is specialized into five types of joinpoints: InnerTransactionJP, OuterTransactionJP, LockingJP, ResourceLockedJP, and OperationJP. *InnerTransactionJP* represents the region of code or period during which a specific transaction code is executed, where advice can be woven in, when *TransJP* occurs after the begin event and before the commit/abort event (prior the end of the transaction execution flow.) *OuterTransactionJP* represents the region of code or period during which a specific transaction code is called, where advice can be woven in, when *TransJP* occurs before the begin event and after the commit/abort event (after the transaction has completed.) Inner/Outer transaction joinpoints have direct access to the target transaction's context, where the woven advices occur before, after or around these joinpoints. They refer to a transaction context concept, i.e., TransactionContext, which contains the relevant transaction information that is delivered at execution time to a proper transaction knowledge, as shown in Fig. 3.

The UMJDT states that every commit or abort event must have a corresponding begin event. In other words, a begin event can exist without a commit or abort event, but not conversely. The events of these kinds of joinpoints are capable of keeping track of transactions that occur in multiple threads within distributed transactions.

LockingJP represents a joinpoint for acquiring the resources used to perform a particular transaction operation. In other words, it represents the region of code or the period during which a specific transaction code region is executed, where advice can be woven in, starting when a begin lock request event is sent to the resource manager/lock manager and ending when the lock request event is granted or refused. The beginning and end of the lock request code are associated with two events that are capable of keeping track of the lock request within the lock context, as shown in Fig. 3. BeginLockEventJP represents an execution point of the code, where advice can be woven into, when LockingJP runs before executing a set lock event for acquiring the specified resource within the lock context associated with the target transaction. EndLockEventJP represents an execution point of the code, where advice can be woven in, when LockingJP runs in place of a set lock event to get the lock that has been granted or refused for the specified resource within the lock context associated with the target transaction.

ResourceLockedJP represents a joinpoint for complete a lock is held. Therefore, it is the region of code or the period during which a specific transaction code region is executed, where advice can be woven in, when a hold event occurs after setting the lock and end before releasing the lock. The demarcation points of the resource locked joinpoint correspond to two events that are capable of tracking the status of locked resources associated with a target transaction within the lock context. HoldEventJP represents an execution point of the code, where advice can be woven in, when a ResourceLockedJP occurs the after executing a set lock event to hold the specified resource within the lock context associated with the target transaction. ReleaseEventJP represents an execution point of the code, where advice can be woven in, when a ResourceLockedJP occurs before executing a release event to unlock the specified resource within the lock context associated with the target transaction.

OperationJP represents a joinpoint for complete a transaction operation. In other words, it represents the region of code or period during which a specific transaction operation code region is executed, where advice can be woven in, when a transaction occurs invoking any transaction method, (i.e., a method is annotated with a transactional annotation) from within the scope of a transaction, which indicates whether a method will be executed within an operation context associated with the target transaction. This joinpoint contains BeforeOperationEventJP and AfterOperationEventJP for keeping track of the status

of all transaction operations. BeforeOperationEventJP represents an execution point of the code, where advice can be woven in, when an OperationJP occurs before executing a reflective access to the information about a transaction operation associated with the target transaction. AfterOperationEventJP represents an execution point of the code, where advice can be woven in, when an OperationJP occurs after executing the reflective access to the state available and information about a transaction operation within the operational context associated with the target transaction.

TransJ can add transactions to the list of possible contexts, which consist of loosely-coupled abstractions. These contexts include various pieces of interesting data and metadata that woven advice might use, e.g., identifier, status, sets of tentative values, rollback logs, snapshots, lock information, timestamps, and other kinds of metadata. Each context includes the location of the joinpoint and runtime information about the transaction objects involved.

Fig. 3 presents the context as a composite class that maintains a collection of sub-contexts in term of a tree structure to represent part-whole hierarchies: Transaction Context, Lock Context, and Operation Context. These contexts represent concrete primitive contexts; for example a lock context or an operation context can be part of a transaction context, which in turn can be part of the parent transaction context.

TransactionContext encapsulates the transaction information that have to be shared among all the outer and inner transaction joinpoints, such as transaction identifier, starting time, commit time, abort time, sub-transactions, status, timestamp, tentative values for resources, etc. LockContext encapsulates the lock information related to underlying resources along with their transactions, such as a locked time, a released time, time-out, status of shared resources, lock mode, lock result, lock owner, tentative values to the update resource, etc. This information has to be available to the LockingJP and ResourcelockedJP. OperationContext encapsulates information about the sequence of the transaction operations in the transaction's body, and operations in progress, etc. This information has to be available to the OperationJP.

TransJ considers events of joinpoints as largely independent, while a context considers them as interrelated through its call transaction concepts that would lead to a more reusable and robust implementation. These events keep track and record the transaction identifying information, e.g., the TID (transaction identifier), for all types of joinpoints. When advice executes, it can access the context information about the joinpoint at which it was invoked. Thus, the *TransJ*'s context is a dynamic and depends on the target transaction objects involved in the joinpoint.

4.2 Registry for Contexts

Transaction aspects dynamically gather context information for all *TransJ* joinpoints. Advices can be executed before, after, or around various *contexts*, which can access joinpoint objects to obtain context information, like a transaction's start time, identifier, status, or the underlying lock information. This task can be facilitated if every context maintains a list of all the transactions that have accessed it. ContextJPRegistry provides this functionality by keeping a list of all transaction-related contexts that have interacted with the target transaction. On this, ContextJPRegistry represents a repository for all transaction-related contexts, which provide relevant information for advices associated with the target transaction at execution time.

For instance, when a joinpoint event occurs, e.g., BeginEventJP, *TransJ* creates an instance of a joinpoint class, e.g., InnerTransactionJP, that further correlates it with other events in the same joinpoint associated with a target transaction, and then adds the instance of the joinpoint to a relevant context, which contains a collection of joinpoints of the target

transaction, and then adds the context to the registry, which contains a collection of contexts, i.e., ContextJPRegistry. When a joinpoint aspect, e.g., InnerOuterTransactionAspect, discovers a relevant transaction joinpoint and correlates it with other appropriate joinpoints that belong to the same transaction as shown in Fig. 3.

4.3 Trackers

Behind the scenes, *TransJ* uses context gathering mechanisms, namely joinpoint tracker aspects, that are based on context-aware transaction abstractions and employed joinpoint events to dynamically gather the relevant information. The trackers work as monitors [19] that perform pattern matching on transaction events, to track individual events and to organize them into high-level transaction-related contexts. Since the monitoring of transactions is itself a crosscutting concern, trackers are implemented as aspects that weave the necessary monitoring logic into places where a transaction event may take place. *TransJ* can support many different kinds of transaction joinpoint trackers, Figs. 4 and 5 show two special types of trackers, namely TransactionJoinPointTracker and ConcurrencyControlJoinpointTracker.

TransactionJoinPointTracker is an aspect that hides transaction-related abstractions in the core transaction application. It crosscuts begin, commit, abort, and transaction operation "@transactional" abstractions and defines a set of elegant and parameterized pointcuts. These provide benefits for sharing states between advices while overcoming the syntactic and semantic variations, defined on standard JTA and Arjuna pre-built libraries, i.e., javax.transaction and com.arjuna.ats.arjuna. These pointcuts are rich enough to encapsulate abstractions for transaction-related concepts of the client and server sides, e.g., UserTransaction and TransactionManager, respectively.

This aspect discovers a relevant joinpoint of the transaction based on the knowledge of access transactions, i.e., access to external transactions or access to internal transactions. Hence, *TransJ* creates seven clean, well-encapsulated transaction-related abstractions for all kinds of types begin, commit, rollback, and transactional annotation (shown in Fig. 4), summarized as follows: Transaction pointcuts for begins: These pointcuts unify syntactic and semantic variations in JTA libraries, i.e., JTA API and Arjuna API, and crosscut outer and inner transaction begin abstractions. Transaction pointcuts for commits and aborts: These pointcuts unify syntactic and semantic variations in JTA libraries, i.e., JTA API, and Arjuna API, and crosscut outer and inner transaction commit and abort abstractions, respectively.

ConcurrencyControlJoinpointTracker is an aspect that hides concurrency control abstractions in core transaction applications. This aspect crosscuts the syntactic and semantic variations exist on standard JTA like pre-built Arjuna library and unifies them into a set of parameterized pointcuts in set lock and release lock abstractions. These pointcuts are rich enough to encapsulate and manage all concurrency-related abstractions and styles related to the locking and unlocking of shared resources in distributed transactions. Hence, *TransJ* provides two clean, well-encapsulated transaction-related abstractions for setlock and doRelease constructs (shown in Fig. 5). These are summarized as follows: Concurrency Control pointcut for setlock: It crosscuts setlock operation for the lock managers in Arjuna API while requesting to hold a specified resource to the associated transaction. Concurrency Control pointcut for doRelease: It crosscuts doRelease operation for the lock manager in Arjuna API while releasing a lock of a specified resource from the associated transaction.

4.4 Base Transaction Aspects

TransJ implements transaction-related crosscutting concerns as aspects derived from transaction aspects that cut through their respective joinpoint trackers.

```
public aspect TransactionJoinPointTracker {
     private Logger logger = Logger.getLogger(TransactionJoinPointTracker.class);

     public pointcut InnerBeginTransaction():
          (execution(* javax.transaction.TransactionManager+.begin())||
               execution(* javax.transaction.UserTransaction+.begin())||
               (execution(* com.arjuna..BaseTransaction+.begin()) &&
                    (!cflowbelow((execution(* javax.transaction.TransactionManager+.begin())||
                                 execution(* javax.transaction.UserTransaction+.begin()))));

     public pointcut InnerCommitTransaction():
          (execution(* javax.transaction.TransactionManager+.commit())||
               execution(* javax.transaction.UserTransaction+.commit())||
               (execution(* com.arjuna..BaseTransaction+.commit()) &&
                    (!cflowbelow((execution(* javax.transaction.TransactionManager+.commit())||
                                 execution(* javax.transaction.UserTransaction+.commit()))));

     public pointcut InnerAbortTransaction():
          (execution(* javax.transaction.TransactionManager+.rollback())||
               execution(* javax.transaction.UserTransaction+.rollback())||
               (execution(* com.arjuna..BaseTransaction+.rollback()) &&
                    (!cflowbelow((execution(*javax.transaction.TransactionManager+.rollback())||
                                 execution(*javax.transaction.UserTransaction+.rollback()))));

     public pointcut OuterBeginTransaction():
          (call(* javax.transaction.TransactionManager+.begin())||
               call(* javax.transaction.UserTransaction+.begin())||
               (call(* com.arjuna..BaseTransaction+.begin()) &&
                    (!cflowbelow((call(* javax.transaction.TransactionManager+.begin())||
                                  call(* javax.transaction.UserTransaction+.begin()))));

     public pointcut OuterCommitTransaction():
          (call(* javax.transaction.TransactionManager+.commit())||
               call(* javax.transaction.UserTransaction+.commit())||
               (call(* com.arjuna..BaseTransaction+.commit()) &&
                    (!cflowbelow((call(* javax.transaction.TransactionManager+.commit())||
                                  call(* javax.transaction.UserTransaction+.commit()))));

     public pointcut OuterAbortTransaction():
          (call(* javax.transaction.TransactionManager+.rollback())||
               call(* javax.transaction.UserTransaction+.rollback())||
               (call(* com.arjuna..BaseTransaction+.rollback()) &&
                    (!cflowbelow((call(* javax.transaction.TransactionManager+.rollback())||
                                  call(* javax.transaction.UserTransaction+.rollback()))));

     public pointcut TransactionMethod(Transactional transaction): execution(@Transactional * *.
*(..)) && @annotation(transaction);
...
}
```

Fig. 4 **A code snippet of TransactionJoinPointTracker.**

```
public aspect ConcurrencyControlJoinPointTracker {
      private Logger logger = Logger.getLogger(ConcurrencyControlJoinPointTracker.class);

      public pointcut SetLock(Lock toSet, int retry, int sleepTime): execution(*
com.arjuna.ats.txoj.LockManager+.setlock(..)) && args(toSet, retry, sleepTime);
      public pointcut DoRelease(Uid id, boolean all): execution(*
com.arjuna.ats.txoj.LockManager+.doRelease(..)) && args(id, all);
...
}
```

Fig. 5 **A code snippet of ConcurrencyControlJoinPointTracker.**

These aspects are derived from abstract *TransactionAspect*, which provides high-level concrete pointcuts that dynamically track different transaction abstractions, as shown in Fig. 6.

The pointcuts in the TransactionAspect take a list of objects as parameters, because this is how concrete aspects based on these pointcuts can access transaction-related context information. We bind context data to pointcut variables, which can then be used to parameterize advices. This allows concrete aspects to be parameterized and configures different joinpoints, which enable reusable aspects to be customized in different contexts and thus increase the reusability of aspects.

The base aspects consist of three distinct abstract aspects correspond to three different kinds of contexts, as mentioned earlier, and extend *TransactionAspect* with pointcut abstractions that are meaningful to those contexts (see Fig. 6). On this, developers can create their own application-level transaction aspects that inherit from these aspects and include advice based on these pointcuts.

InnerOuterTransactionAspect extends *TransactionAspect* with pointcuts for transaction beginnings and the transaction ends as shown in Fig. 7. It involves begin, commit and abort joinpoints to demarcate the transaction scope. It defines six pointcuts: iTransactionBegin[1], iTransactionCommit, iTransactionAbort, oTransactionBegin[2], oTransactionCommit and oTransactionAbort. These pointcuts crosscut TransactionJoinpointTracker to establish a transaction context on the client application and the application server sides of each executed transaction. The oTransactionBegin creates an OuterTransactionJP and instantiates a transaction context. The oTransactionCommit or oTransactionAbort retrieves the matching OuterTransactionJP from the target TransactionContext in ContextJPRegistry and ends a transaction after a client or transaction manager invokes a commit or abort joinpoint event. The iTransactionBegin creates an InnerTransactionJP and starts a transaction when a client or transaction manager executes a begin event, and then retrieves the matching target TransactionContext from the ContextJPRegistry and adds the InnerTransactionJP. The iTransactionCommit or iTransactionAbort retrieves the matching InnerTransactionJP from the target TransactionContext in the ContextJPRegistry and adds the commit joinpoint event or abort joinpoint event, respectively, as shown in Fig. 8. Developers can use this kind of aspect to weave advice before, after, or around entire transactions, either from a transaction application client or application server perspective in different transaction models (flat or nested).

OperationAspect extends *TransactionAspect* with pointcuts for transaction operation as shown in Fig. 9. They provide a way for applications to capture arbitrarily complex operations, which define the sequence of transaction operations that comprise the transaction body. This aspect defines pointcuts to demarcate the transaction operation scope, namely BeforeTransactionOperation and AfterTransactionOperation. The BeforeTransactionOperation creates an OperatoinJP and instantiates an OperationContext as shown in Fig. 10. It exposes the before-operation event joinpoint to the OperationJP and then adds the OperationContext to the ContextJPRegistry. The AfterTransactionOperation retrieves the matching OperationJP from the OperationContext for the current transaction in the ContextJPRegistry, and exposes the after-operation event joinpoint to the OperationJP. Developers can use this aspect to weave advice before, after, or around a transaction operation.

LockAspect is derived from the TransactionAspect and thereby inherits the locking and resource-locked

[1] The transaction pointcut is initialized with a lowercase letter that indicates where the joinpoint is located, (i) stands for inner transaction; iTransactionBegin means inner begin transaction.

[2] The transaction pointcut is initialized with a lowercase letter that indicates where the joinpoint is located, (o) stands for outer transaction; oTransactionBegin means outer begin transaction.

```
public abstract aspect TransactionAspect
{
      private Logger logger = Logger.getLogger(TransactionAspect.class);

      protected pointcut InnerTransactionBegin(Xid tid, ..):
            execution(void TransactionJoinPointTracker.innerTransactionBegin(..)) && args(tid,..);

      protected pointcut InnerTransactionCommit(Xid _tid, ..):
            execution(void TransactionJoinPointTracker.innerTransactionCommit(..)) && args(_tid, ..);

      protected pointcut InnerTransactionAbort(Xid _tid, ..):
            execution(void TransactionJoinPointTracker.innerTransactionAbort(..)) && args(_tid,..);

      protected pointcut OuterTransactionBegin(): execution(void TransactionJoinPointTracker.outerTransactionBegin(..));

      protected pointcut OuterTransactionCommit():execution(void TransactionJoinPointTracker.outerTransactionCommit(..));

      protected pointcut OuterTransactionAbort(): execution(void TransactionJoinPointTracker.outerTransactionAbort(..));

      protected pointcut BeginlockOperation(Xid tid, ..):
            execution(void ConcurrencyControlJoinPointTracker.beginlockOperation(..)) && args(tid,..);

      protected pointcut EndlockOperation(String lockResult, Uid lockId, Uid lockOwnerId,..):
            execution(void ConcurrencyControlJoinPointTracker.endlockOperation(..)) && args(lockResult, lockId, lockOwnerId,
..);


      protected pointcut SetHold(Xid tid, Uid lockId, Uid lockOwnerId, ..):
            execution(void ConcurrencyControlJoinPointTracker.setHold(..)) && args(tid, lockId, lockOwnerId,..);

      protected pointcut ReleaseHold(Uid lockId, ..):
            execution(void ConcurrencyControlJoinPointTracker.releaseHold(..)) && args(lockId, ..);


      protected pointcut BeforeOperation(Method method, Xid tid, ..):
            execution(void TransactionJoinPointTracker.beforeTransactionMethod(..)) && args(method, tid,..);

      protected pointcut AfterOperation(String operationName, Object [] arguments, ..):
            execution(void TransactionJoinPointTracker.afterTransactionMethod(..)) && args(operationName, arguments,..);
      ...
```

Fig. 6 **A code snippet of TransactionAspect.**

```
public abstract aspect InnerOuterTransactionAspect extends TransactionAspect{
     private Logger logger = Logger.getLogger(InnerOuterTransactionAspect.class);

     public pointcut oTransactionBegin(OuterTransactionJP _outerTransactionJp) :
execution(* InnerOuterTransactionAspect+.Begin(OuterTransactionJP)) &&
args(_outerTransactionJp);

     public pointcut oTransactionCommit(OuterTransactionJP _outerTransactionJp):
execution(* InnerOuterTransactionAspect+.Commit(OuterTransactionJP)) &&
args(_outerTransactionJp);

     public pointcut oTransactionAbort(OuterTransactionJP _outerTransactionJp) :
execution(* InnerOuterTransactionAspect+.Abort(OuterTransactionJP)) &&
args(_outerTransactionJp);

     public pointcut iTransactionBegin(InnerTransactionJP _innerTransactionJp) :
execution(* InnerOuterTransactionAspect+.Begin(InnerTransactionJP)) &&
args(_innerTransactionJp);

     public pointcut iTransactionCommit(InnerTransactionJP _innerTransactionJp):
execution(* InnerOuterTransactionAspect+.Commit(InnerTransactionJP)) &&
args(_innerTransactionJp);

     public pointcut iTransactionAbort(InnerTransactionJP _innerTransactionJp) :
execution(* InnerOuterTransactionAspect+.Abort(InnerTransactionJP)) &&
args(_innerTransactionJp);
     ...
}
```

Fig. 7 **Extended parametrized-pointcuts in InnerOuterTransactionAspect.**

```
public abstract aspect InnerOuterTransactionAspect extends TransactionAspect{
	...
	// outer transaction scope
	before(): OuterTransactionBegin() {
		outerTransactionjp = new OuterTransactionJP();
		BeginEventJP beginEventJP = new BeginEventJP();
		...
		outerTransactionjp.setBeginEventJP(beginEventJP);
		transactionContext = new TransactionContext();
		...
		ContextJPRegistry.getInstance().addContext(transactionContext);
		...
	}

	after() : OuterTransactionCommit() {
		CommitEventJP commitEventjp = new CommitEventJP();
		...
		((OuterTransactionJP)ContextJPRegistry.getInstance().findContext(outerTransactionjp).getTransJp()).setCommitEventJP(commitEventjp);
		...
	}

	after() : OuterTransactionAbort() {
		AbortEventJP abortEventjp = new AbortEventJP();
		...
		((OuterTransactionJP)ContextJPRegistry.getInstance().findContext(outerTransactionjp).getTransJp()).setAbortEventJP(abortEventjp);
		...
	}

	// inner transaction scope
	after(..) : InnerTransactionBegin(..) {
		...
		innerTransactionjp = new InnerTransactionJP(tid);
		BeginEventJP beginEventJP = new BeginEventJP(tid);
		...
		beginEventJP.setBeginTime(beginTimestamp);
		innerTransactionjp.setBeginEventJP(beginEventJP);
		...
		((TransactionContext)ContextJPRegistry.getInstance().findContext(contextUid)).setTid(tid);;
		...
		((TransactionContext)ContextJPRegistry.getInstance().findContext(contextUid)).setInnerJP(innerTransactionjp);
		...
	}

	before(..) : InnerTransactionCommit(..) {
		CommitEventJP commitEventjp = new CommitEventJP();
		((InnerTransactionJP)ContextJPRegistry.getInstance().findContext(innerTransactionjp).getTransJp()).setCommitEventJP(commitEventjp);
		innerTransactionjp.setCommitEventJP(commitEventjp);
		...
	}

	before(..) : InnerTransactionAbort(..) {
		AbortEventJP abortEventjp = new AbortEventJP();
		((InnerTransactionJP)ContextJPRegistry.getInstance().findContext(innerTransactionjp).getTransJp()).setAbortEventJP(abortEventjp);
		...
	}
	...
}
```

Fig. 8 A code snippet of InnerOuterTransactionAspect.

```
public abstract aspect TransactionOperationAspect extends TransactionAspect{
     private Logger logger = Logger.getLogger(TransactionOperationAspect.class);

     declare parents: TransactionOperationAspect extends TransactionAspect;

     public pointcut BeforeTransactionOperation(OperationJP _operationjp) :
execution(* TransactionOperationAspect+.beforeOperation(OperationJP)) &&
args(_operationjp);
     public pointcut AfterTransactionOperation(OperationJP _operationjp):
execution(* TransactionOperationAspect+.afterOperation(OperationJP)) &&
args(_operationjp);
...
}
```

Fig. 9 Extended parameterized-pointcuts in TransactionOperationAspect.

```
public abstract aspect TransactionOperationAspect extends TransactionAspect{
      ...
      before(..) : BeforeOperation(..) {

            operationjp = new OperationJP();
            ...
            BeforeOperationEventJP beforeEventJP = new BeforeOperationEventJP();
            ...
            operationjp.setBeforeOperationEventjp(beforeEventJP);
            ...
            operationContext = new OperationContext();
            operationContext.setOperationjp(operationjp);
            ...
            ContextJPRegistry.getInstance().addContext(operationContext);
      }

      after(..) : AfterOperation(..){

            AfterOperationEventJP afterEventJP = new AfterOperationEventJP();
            ...
            ((OperationJP)
ContextJPRegistry.getInstance().findContext(operationjp).getTransJp()).setAfterOperationEventjp(afterEventJP);
            ...
            operationjp.setAfterOperationEventjp(afterEventJP);
      }
      ...
}
```

Fig. 10 A code snippet of TransactionOperationAspect.

pointcuts, as shown in Fig. 11. It involves setlock-event and release-event joinpoints to associate and disassociate the specified resource to/from the target transaction. It defines pointcuts BeginRequestlock, EndRequestlock, HoldingResource, and ReleasingResource that crosscut ConcurrencyControlJoinpointTracker to establish the lock context. BeginRequestlock creates an instance of LockingJP, exposes BeginlockEventJP to it, instantiates a lock context, and then adds the context to the ContextJPRegistry, as shown in Fig. 12. The EndRequestlock retrieves the matching LockingJP from the target LockContext in the ContextJPRegistry, and exposes the EndlockEventJP to the LockingJP when the request lock is granted or refused. The HoldingResource creates a ResourceLockedJP and exposes the hold-event joinpoint to it. It also retrieves the matching lock context from ContextJPRegistry and then adds the ResourceLockedJP to the lock context in ContextJPRegistry. The ReleasingResource retrieves the matching ResourceLockedJP from the target LockContext in the ContextJPRegistry and then exposes the release-event joinpoint to the ResourcelockedJP and ends the locked resource. Developers can use this aspect to weave advice before, after, or around the entire locking perspective.

In DTPSs, the nested and concurrent transactions may occur with multiple other hosts, i.e., transaction in progress, which are also involved in a multi-threaded process. The aspects can apply for a transaction and keep track of the multiple concurrent transactions by maintaining a collection of contexts. A context for each transaction is maintained in terms of its own current context and association with the in-progress transaction.

5. Reusable and Application-Level Aspects

A developer can implement crosscutting concerns, define transaction-related pointcuts, and weave advice into any of above joinpoints by specializing the corresponding abstract *TransJ* aspects that are shown in yellow boxes in Fig. 3. *TransJ* implementation provides generic advices in the base aspects that follow the template method pattern [20]. Therefore, the base aspects are implemented as abstract aspects to contain the actual implementation of the template advices and pointcuts. This allows developers to quickly adapt them to the specific needs of their application by reusing and integrating them into existing or new applications. *TransJ* has to contain a

```
public abstract aspect LockAspect extends TransactionAspect {
          private Logger logger = Logger.getLogger(LockAspect.class);

          public pointcut BeginRequestlock(LockingJP _lockingjp): execution(*
LockAspect+.beginSetLocking(LockingJP)) && args(_lockingjp);

          public pointcut EndRequestlock(LockingJP _lockingjp): execution(*
LockAspect+.endSetLocking(LockingJP)) && args(_lockingjp);

          public pointcut HoldingResource(ResourceLockedJP _resourceLockedjp):
execution(* LockAspect+.hold(ResourceLockedJP)) && args(_resourceLockedjp);

          public pointcut ReleasingResource(ResourceLockedJP _resourceLockedjp):
execution(* LockAspect+.release(ResourceLockedJP)) && args(_resourceLockedjp);
     ...
}
```

Fig. 11 Extended parameterized-pointcuts in LockAspect.

```
public abstract aspect LockAspect extends TransactionAspect {

          ...

          before(..) : BeginlockOperation(..) {
               lockingjp = new LockingJP();
               BeginLockEventJP beginlockEventjp = new BeginLockEventJP();
               ...
               lockingjp.setBeginlockEventjp(beginlockEventjp);
               ...
               lockContext = new LockContext();
               ...
               lockContext.setLockingjp(lockingjp);
               ContextJPRegistry.getInstance().addContext(lockContext);
          }

          after (..) : EndlockOperation(..) {
               EndLockEventJP endlockEventjp = new EndLockEventJP();
               ...
               endlockEventjp.setElapsedTime(elapsedTime);
               ...
               ((LockingJP)
ContextJPRegistry.getInstance().findContext(lockingjp).getTransJp()).setEndlockEven
tjp(endlockEventjp);

               lockingjp.setEndlockEventjp(endlockEventjp);
               ...
          }

          after(..): SetHold(..) {
               resourceLockedjp = new ResourceLockedJP();
               HoldEventJP holdEventJP = new HoldEventJP(lockOwnerId);
               ...
               resourceLockedjp.setHoldEventjp(holdEventJP);
               ...
               ((LockContext)
ContextJPRegistry.getInstance().findContext(lockingjp)).setResourceLockedjp(resourc
eLockedjp);
               lockContext.setResourceLockedjp(resourceLockedjp);
               ...
          }

          before(..) : ReleaseHold(..) {
               ...
               ReleaseEventJP releaseEventjp = new ReleaseEventJP();
               ...
               ((ResourceLockedJP)
ContextJPRegistry.getInstance().findContext(resourceLockedjp).getTransJp()).setRele
aseEventjp(releaseEventjp);
               resourceLockedjp.setReleaseEventjp(releaseEventjp);
               ...
          }
```

Fig. 12 An aspect code snippet of Lock Aspect.

reusable aspect library that supports AO features derived from the base aspects. As an example, this section describes the implementation of a reusable and an application-level aspect that weaves performance measurements in the distributed transaction applications.

Aspect developers implement reusable aspects by specializing the base aspects in *TransJ*. The reusable aspects represent general crosscutting concerns commonly found in distributed applications with significant transaction requirements as mentioned above. Table 1 lists the aspects currently in the reusable aspects library and Fig. 13 shows part of the implementation of one of them. This is called *Total Turn Around Time Monitor*. *TransJ* provides a library of reusable aspects for transaction-related crosscutting concerns, like *TransctionTurnAroundTime*, that helps programmers measure the responsiveness time. These aspects allow programmers to adapt the reusable aspects to new demands and to cope with the specific needs of their application by overriding these methods. *TransJ* library offers other reusable aspects the make use of this and other reuse techniques to integrate them easily into existing or new applications. We expect that reusable aspects will continue to grow as new generally-applicable transaction aspects are discovered, implemented, and documented.

For discussion purposes, assume that the performance measurements are a throughput and an average-transaction response turnaround time statistic. *TransctionTurnAroundTime* is an extension that measures some performance-related statistics for transactions between a client and application server (e.g., JBoss Application Server). Also, assume that the core application considers a transaction to be the completion a set of sub-transactions. Consider a transaction involving three sub-transactions. So, we can measure throughput for a unit of time, 60 seconds, by simply counting the number of these transactions completed in that period. The average response turnaround time is the average of time spans from transaction begin times to transaction commit or abort times.

First, notice how this advice is derived from InnerOuterTransactionAspect and in doing so, it can reuse its implementation of the transaction turnaround time concept directly. Fig. 14 shows a snippet of code that presents the implementation of measuring the total turnaround time and throughput for a nested transaction at the application level. As mentioned, the developers can implement and add application-level aspects into core application logic by reusing reusable aspects or extending base aspects in *TransJ*. Second, notice how the aspect is derived from Transaction Turn

Table 1 Sample reusable crosscutting concerns in transactions.

Aspect name	Description
Optimizer	Tracks workload based on the most likely behaviour of a transaction. It helps programmers to determine the most efficient way to execute a transaction by considering the possible behaviours on shared resources.
Performance anylazer	Helps the programmer to analyze the vast amount of transaction resource accesses for improving the application performance.
Notification	Allows the developer to activate alarms for critical errors, exceptions, time-based expiration, and invalid state.
Authenticator	Tracks consistent and secure transactions for handling authentication permissions in transaction domain.
Audit trail	Records a history of actions executed by transactions and users. It includes a chronological list of steps that are required in order to begin a transaction, as well as bring it to completion. It records information such as who has accessed a transaction, what operation was performed on it, when it was performed, and how the state was changed.
Logging By transaction	Logs transaction context operations in a developer-defined format and domain.
JustIntime	Provides virtual helper methods for a transaction which help programmers share context information across hosts when necessary.
Total turn around time monitor	Provides virtual helper methods for transactions which help programmers measure the responsiveness time by overriding their aspects in application level.

```
public abstract aspect TransactionTurnArounTime extends
InnerOuterTransactionAspect{

     protected pointcut beginTransaction(TransactionContext _transactionContext)
: execution(* TransactionTurnArounTime+.begin(TransactionContext)) &&
args(_transactionContext);
     protected pointcut endTransaction(TransactionContext _transactionContext) :
execution(* TransactionTurnArounTime+.end(TransactionContext)) &&
args(_transactionContext);

     public Timestamp TransactionContext.beginTime =null;
     public long TransactionContext.turnAroundTime=0;


     before(InnerTransactionJP innerjp): iTransactionBegin(innerjp)
     {
          TransactionContext transactionContext = (TransactionContext)
ContextJPRegistry.getInstance().findContext(innerjp);
          transactionContext.beginTime = new Timestamp();
          begin(transactionContext);
     }

     after(InnerTransactionJP innerjp) : iTransactionCommit(innerjp) ||
iTransactionAbort(innerjp)
     {
          TransactionContext transactionContext = (TransactionContext)
ContextJPRegistry.getInstance().findContext(innerjp);
          transactionContext.turnAroundTime = ((new Timestamp()).getMillSeconds()
- transactionContext.beginTime.getMillSeconds())/1000;
          end(transactionContext);
     }

     private pointcut contextCreation(TransactionContext context):
     target(context) && execution(public TransactionContext+.new(..)) ;

     after(TransactionContext context) : contextCreation(context)
     {
     }

     protected void begin(TransactionContext _transactionContext){
     }

     protected void end(TransactionContext _transactionContext){
     }
}
```

Fig. 13 A reusable-aspect code snippet of TransactionTurnAroundTime.

```
public aspect MyAppPerformanceMonitor extends TransactionTurnAroundTime{

     private int transactionCount =0;
     private long totalAroundTime =0;
     private Stats stat = new Stats();

     @Override
     protected void end(TransactionContext transactionContext){

          int i =0;
          totalAroundTime += transactionContext.turnAroundTime;
          // Number of completed transactions
          transactionCount++;

          stat.computeAvgResponseTime(totalAroundTime, transactionCount);
     }
}

public class Stats{

     private long completeTransactionCount =0;
     private float avgResponseTime =0f;

     public void computeAvgResponseTime(long turnAroundTime, int accessCount){
          //Efficiency per minute
          avgResponseTime = (float)(turnAroundTime/accessCount);
     }
}
```

Fig. 14 Performance measure crosscutting concern.

Around Time aspect and in doing so, it can reuse its implementation of the transaction turnaround time concept directly. Then, it adds some additional behavior at the end of a transaction to compute the average of the transaction responsiveness time per-minute (60 seconds), i.e., efficiency.

6. Experimental Hypotheses

The theoretical ideas that underpin *TransJ* lead to the eight concrete hypotheses. All of these hypotheses have the same premise and refer to the metrics that will be defined in a new extended quality model for distributed applications. To determine whether *TransJ* improves reusability without sacrificing performance, the authors will conduct an experiment that tests the following hypotheses:

(1) If transaction-related crosscutting concerns are effectively modularized and encapsulated in *TransJ* aspects, then the software has better encapsulation and SoCs (Separation of Concerns) and less scattering than equivalent systems developed with AOP design techniques, especially AspectJ;

(2) If transaction-related crosscutting concerns are effectively encapsulated in *TransJ* aspects, then the software has a lower coupling than equivalent systems developed with AOP design techniques, especially AspectJ;

(3) If transaction-related crosscutting concerns are effectively encapsulated in *TransJ* aspects, then the software has higher cohesion and less tangling than equivalent systems developed with AOP design techniques, especially AspectJ;

(4) If transaction-related crosscutting concerns are effectively encapsulated in *TransJ* aspects, then the software is not significantly larger or complex than equivalent systems developed with AOP design techniques, especially AspectJ;

(5) If transaction-related crosscutting concerns are effectively encapsulated in *TransJ* aspects, then the software is significantly more oblivious than equivalent systems developed with AOP design techniques, especially AspectJ;

(6) If transaction-related crosscutting concerns are effectively encapsulated in *TransJ* aspects, then the software would preserve or improve runtime performance compared with equivalent systems developed with AOP design techniques, especially AspectJ;

(7) If transaction-related crosscutting concerns are effectively encapsulated in *TransJ* aspects, then the extension part, i.e., crosscutting concern, of the software would require a smaller number of changes to reuse compared to equivalent systems developed with AOP design techniques, especially AspectJ;

(8) If *TransJ* provides a better modularization of transaction-related crosscutting concerns, then the development transaction system would be less complicated and more readable. Thus, software development efficiency would be increased, so that the system would be created faster than equivalent systems developed with AOP design techniques. In other words, the total programmer's working time should be shorter than the development time of analogous systems developed with AOP techniques, especially AspectJ.

For the moment, proposing the complete implementation of an extension to AspectJ that performs the expected weaving and tracks transaction-related context information—the abstract independent framework for basic weaving into high-level abstractions—are sufficiently interesting and potentially beneficial to dominate our immediate attention. Our next steps are to test the stated hypotheses through performing a preliminary experiment to validate whether each hypothesis is true or false.

7. Related Work

All ideas, concepts and approaches investigated in this section intersect with a broad spectrum of research projects on transactional systems, transactional aspects, reusable AO frameworks,

application-level aspects and interactions. We offer some relevant research in this section.

Kiczales et al. [3], introduced the idea of weaving logic for crosscutting concerns into core applications was introduced over 15 years ago, and their work stems from even earlier research with inheritance, aggregation, and mix-ins [2]. Like all great ideas, the heart of the weaving solution is relatively straightforward—modularize concerns into first-class constructs, find the right place(s) to introduce appropriate logic from those constructs, and the either insert code that executes the new logic unconditional (because it can be determined to always be needed) or insert code that makes a final decision about executing the new code at runtime. Raza et al., present the design and implementation of a new AOPL framework, called CommJ, which is an extension to AspectJ for enabling programmers to encapsulate communication-related crosscutting concerns in modular, cohesive and loosely coupled aspects [21]. CommJ allows developers to weave crosscutting concerns into IPC (inter-process communications) in a modular and reusable way, while keeping the core functionality oblivious to those concerns. This is in many respects, we found some conceptual similarity with this design approach to our work, but we have a different goal in that it addresses how to weave transaction-related crosscutting concerns into high-level runtime abstractions, i.e., distributed transactions. Additionally, *TransJ* framework provides low-level distributed aspects that perform the expected weaving and track of context information. We believe this to be feasible because it is similar to the technique used by CommJ to add communication-related aspects to AspectJ. It also clearly defines transaction primitives for the DTPS that defines interesting joinpoints relative to transaction execution and related contexts for the woven logic of crosscutting concerns. Therefore, we believe our work that is concerned to pave the way for the weaving of crosscutting concerns into high-level program abstractions that span multiple threads of execution and may be interleaved with concurrent execution of similar the abstraction. It also can define more reusable aspects, which not only can be extended, but can also be combined to build more complex types of transaction concerns.

Implementing transaction-related crosscutting functionality do not fit efficiently into OOP, which leads to an unnecessary code duplication, a complex code, a decrease in software quality, and an increase in product errors and bugs [40]. Several research works are currently underway to explore the feasibility of AOP techniques to deal with the concepts of transaction in various scenarios [5], such as Ref. [37]. The case study proposed in Ref. [24], promises to be a perfect candidate that may serve as a benchmark for evaluating the new AOP approaches, the expressivity of AOP languages, the performance of AOP environments, and the suitability of AO modeling notations. This research presents a language independent decomposition of the ACID (Atomicity, Consistency, Isolation and Durability) properties of transactions into a set of fine-grained aspects, i.e., base aspects, each one providing a well-defined reusable functionality. It then shows how these aspects can be configured and composed in different ways to achieve various concurrency control and recovery strategies for the transactional object. Therefore, this framework enables the design of various concurrency control and recovery concerns through the configuration and composition of these new aspects. However, other concerns, such as transaction life-cycle management, are only primarily supported and badly modularized, and as a result, their functionality cut through the design of the other aspects of the framework. Motivated by these, Kienzle was taken the case study one step further. In Refs. [23], he presents a language-independent framework that provides the runtime support for transactions, called *AspectOptima*. It uses AO technology to decompose transaction models and their implementations into

many individual reusable aspects. In other words, it consists of a collection of ten base aspects that can be configured to guarantee the ACID properties for the transactional object [33]. However, this purpose of this research is not produced implementations for a specific transaction standard like JTA, or a reference implementations.

In comparing with AspectOptima, the *TransJ* discusses the composition of transaction abstractions by separating out the definition of transactions from the definition of other aspects using general-purpose abstract transaction concepts, i.e., high-level abstractions, each one providing a well-defined reusable functionality. We believe our work enables better reuse, encapsulation and obliviousness for transaction-related crosscutting concerns. On the other hand, *TransJ* serves for a more practical purpose. It provides a transaction joinpoint model, which is not the only contribution of this research. In addition, it proposes a new context concept to act as meta-data model for encapsulating the transaction-related information. Furthermore, the research adds new abstract concepts, which correspond to the transaction primitives in JTA and AspectJ frameworks, enable the design of various application-level aspects through the configuration and composition of reusable and base aspects. In short, *TransJ* the only work required by developers to acquire the functionality provided by application-level aspects are to bind their application classes to the appropriate aspects. This requires no knowledge of the inner working of *TransJ*.

In Ref. [25], Sadat-Mohtasham provides a design, and implement *Transactional Pointcuts* as a realization of the new model in the AspectJ language. He also proposes a new joinpoint model, based on the pointcut-advice model and a new construct, *transcut*, which selects sets of interrelated joinpoints and reifies them into higher-level joinpoints that can be advised. The authors have extended abc's existing joinpoint matching infrastructure for transcut matching by implementing the appropriate subclasses (for the new type of shadow, new pointcuts, etc.,) and by advising the right joinpoints to adapt the behavior of some of the existing components in the context of transcut matching. If a transcut matches a shadow, an advice application object is created to be applied to the shadow in the weaving phase. All three major types of advice (i.e., around, after, and before) are supported for transcuts. There are some differences between transactional pointcut model and our work. Transactional pointcut relies on static analysis only and, therefore, is inherently imprecise. Our model uses an interval joinpoints (execution-time joinpoint model) to determine when an advice should stratify. Also, the *TransJ* designation and advice model complies with the existing dynamic pointcut-advice model in AspectJ, which made it possible for integration and interaction with an AOPL, such as AspectJ.

On the other hand, the authors discuss dynamic meta-model annotations to add well-separated concerns. The authors share some design similarities for *TransJ* that is joinpoints in transcut is identified as part of a bigger context and in relation to other joinpoints. *TransJ* design principles include a similar concept for implementing transaction patterns using AspectJ, but the *TransJ* handles transactions in high-level transaction abstractions rather than low-level abstractions. It allows to encapsulate the transaction concerns from core application functionality with writing reusable and application-level transaction aspects as explained in *TransJ*. In addition, it already provides a set of reusable aspects and has the ability to compose and configure the application-level aspects.

8. Conclusion and Future Work

This paper took the necessary steps to introduce the notation of transaction-aware aspects to incorporate transaction-related crosscutting concerns into an AspectJ framework, namely *TransJ*. *TransJ* is an independent abstraction framework that uses aspects as main abstractions and proposes a model for

distributed transaction aspects and transaction joinpoints for weaving crosscutting concerns into transaction abstractions. Thus, it allows developers to encapsulate transaction-related crosscutting concerns in reusable modules, i.e., the reusable library that consists of reusable transaction aspects and doubles as a proof of concepts, since these aspects can be directly applied to a wide range of existing transaction applications. It then shows how these reusable aspects can be configured and composed in different ways to encapsulate new concerns at application-level.

We believe that *TransJ* is capable of encapsulating a wide range of transaction-related crosscutting concerns in aspects. We hope to get empirical evidence of the *TransJ*'s value by increasing the number of aspects in the reusable aspects and by continuing to expand the number and types of applications that use *TransJ*.

Our next steps are to perform a preliminary experiment that we hope will provide evidence of improvement in reuse without sacrificing the performance. To measure the reuse and performance, we will define an extension to existing quality models to be adapted for transactional applications with following new factors: Understandability, Extensibility, Localization of Design Decisions, Obliviousness, Efficiency, Predictability, and Scalability. Each factor is related to well-established software-engineering principles: Separation of Transaction Concerns, Coupling (dependency), Cohesion, Code size and Complexity, Tangling, Scattering, Aspects/Obliviousness, Throughput, Transaction Volume, Transaction Velocity, and Productivity. In order to achieve this as an ambitious goal, we plan to setup an experiment methodology, involving eight quality hypotheses and data collection from the extended quality model for transactional application. We hope the results of the preliminary investigation will provide sufficient evidence on hoped-for benefits to verify hypotheses. Hence, we can conclude that *TransJ* is capable of encapsulating a wide range of transaction-related crosscutting concerns and that it can provide better reusability to refine the core *TransJ* Infrastructure, increase the number of aspects in the reusable layer and continue to expand the number and types of applications that use *TransJ*.

References

[1] Brooks, F. P. Jr. 1987. "No Silver Bullet, Essence and Accidents of Software Engineering." *IEEE Computer* 20 (4): 10-9.

[2] Booch, G., and Maksimchu, R. M. 2007. "Object-Oriented Analysis and Design with Applications." *Addison-Wesley Professional*, third edition. Boston: Published April 1st.

[3] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J. M., and Irwin, J. 1997. "Aspect-Oriented Programming." In *Proceedings of ECOOP '97*, Springer Verlag, 220-42.

[4] Gradecki, J., and Lesiecki, N. 2003. "Mastering AspectJ." Wiley Publishing Inc., 456. ISBN: 978-0-471-43104-6.

[5] Rosenhainer, L. 2004. "Identifying Crosscutting Concerns in Requirements Specifications." In *Proceedings of OOPSLA Early Aspects*. October 2004, Vancouver, Canada. http:// trese.cs.utwente.nl/Docs/workshops/oopsla-early-aspects-2004/.

[6] Clifton, C., and Leavens, G. T. 2003. "Obliviousness, Modular Reasoning, and the Behavior Subtyping Analogy." In *Proceedings of the Workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT)*, Workshop at AOSD, 1-6.

[7] Kiczales, G., and Mezini, M. 2005. "Aspect-Oriented Programming and Modular Reasoning." In *Proc. 27th Int. Conf. Software Engineering*, St. Louis, MO, 49-58.

[8] Eclipse. 2015. AspectJ. [Online]. Available http://www.eclipse.org/aspectJ/.

[9] Aspectwerkz2. 2015. Plain Java AOP. [Online]. Available http://aspectwerkz.codehaus.org/.

[10] Jboss. 2015. JBoss AOP. [Online]. Available http://www.jboss.org/jbossaop.

[11] Spring. 2015. Spring AOP. [Online]. Available http://www.springframework.org.

[12] Gray, J. 1981. "The Transaction Concept: Virtues and limitations." In *Proceedings of the 7th International Conference on VLDB Systems*, ACM, New York, 144-54.

[13] Gray, J., and Reuter, A. 1993. "Transaction Processing: Concepts and Techniques." The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, San Mateo, CA, 1993. ISBN-13: 978-1558601901,

ISBN-10: 1558601902.

[14] Kohad, G., Gupta, S., Gangakhedkar, T., Ahirwar, U., and Kumar, A. 2013. "Concept and Techniques of Transaction Processing of Distributed Database Management System." *International Journal of Computer Architecture and Mobility* 1 (8). ISSN 2319-9229.

[15] AlSobeh, A., and Clyde, S. 2014. "Unified Conceptual Model for Joinpoints in Distributed Transactions." *ICSE'14*. *The Ninth International Conference on Software Engineering Advances*. Nice, France. ISBN: 978-1-61208-367-4.

[16] Wikipedia. 2015. Block Diagram. [Online]. Available http://en.wikipedia.org/wiki/Block_diagram.

[17] Shaw, M., and Garlan, D. 1996. "Software Architecture: Perspectives on an Emerging Discipline." *Upper Saddle River*. NJ: Prentice-Hall.

[18] Sun Microsystems Inc. 1999. Java Transaction API (JTA) [Online]. 901 San Antonio Road, Palo Alto, CA 94303. Available https://www.progress.com/jdbc/resources/tutorials/understanding-jta.

[19] Douence, R., Le Botlan, D., Noyé, J., and Südholt, M. 2006. "Concurrent Aspects." In *Proc. 5th Int. Conf. GPCE*, Portland, OR, 79-88.

[20] Gamma, E., Helm, R., Johnson, R., Vlissides, J., and Booch, G. 1995. *Design Patterns-Elements of Reusable Object Oriented Software*, 3rd edition. Addison-Wesley.

[21] Raza, A., and Clyde, S. 2013. "Weaving Crosscutting Concerns into Inter-process Communications (IPC) in AspectJ." *ICSEA 2013*, Venice, Italy, 234-40. ISBN: 978-1-61208-304-9.

[22] Redhat. 2007. Jboss Transaction Manager. [Online]. Available http://docs.jboss.org/jbosstm/docs/4.2.3/manuals/pdf/jta/ProgrammersGuide.pdf.

[23] Rashid, A., and Chitchyan, R. 2003. "Persistence as an Aspect." In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development-Development (AOSD)*, Boston, MA, USA, 120-9. ACM, New York, NY, USA.

[24] Kienzle, J., and G´elineau, S. 2006. "Ao Challenge-Implementing the Acid Properties for Transactional Objects." In *AOSD '06*: *Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, 202-13, New York, NY, USA. ACM Press.

[25] Sadat-Mohtasham, H., and Hoover, H. S. 2009. "Transactional Pointcuts: Designation Reification and Advice of Interrelated Join Points." In *Proceedings of the 8th International Conference on Generative Programming and Component Engineering*, 35-44. ACM.

[26] Rausch, A., Rumpe, B., Klein, C., and Hoogendoom, L. 2003. "Aspect-Oriented Framework Modeling." *4th AOM Workshop at UML'03*, San Francisco, CA.

[27] Kienzle, J., and Guerraoui, R. 2002. "AOP: Does it Make Sense? The Case of Concurrency and Failures." *Proc. ECOOP'02*, 37-61.

[28] Kamble, G. 2009. "Aop-Introduced Crosscutting Concerns." *Proceedings of 2009 International Symposium on Computing*, Communication, and Control. Singapore, 140-4. ISBN 978-9-8108-3815-7.

[29] Raza, A., and Clyde, S. 2014. "Communication Aspects with CommJ: Initial Experiment Show Promising Improvements in Reusability and Maintainability." *ICSEA 2014*. Nice, France.

[30] Akai, S., and Chiba, S. 2009. "Extending AspectJ for Separating Regions." In *GPCE*, 45-54. ACM.

[31] Alkhatib, G., and Labban, R. S. 1995. "Transaction Management in Distributed Database Systems: The Case of Oracle's Two-Phase Commit." *The Journal of Information Systems Education* 13 (2): 95-103.

[32] Härder, T., and Rothermel, K. 1993. "Concurrency Control Issues in Nested Transactions." *Journal of VLDB* 2 (1): 39-74.

[33] Kienzle, J., Duala-Ekoko, E., and Gelineau, S. 2009. "Aspect Optima: A Case Study on Aspect Dependencies and Interactions." *Trans Aspect-Oriented Softw Dev* 5: 187-234. Berlin, Heidelberg: Springer-Verlag. doi:10.1007/978-3-642-02059-9_6.

[34] Sirbi, K., and Kulkarni, P. J. 2010. "Enhancing Modularity in Aspect-Oriented Software Systems—An Assessment Study." *(IJCSE) International Journal on Computer Science and Engineering* 2 (6).

[35] Sant'Anna, C., Lobato, C., Kulesza, U., and Lucena, C. 2008. "On the Modularity Assessment of Aspect-Oriented Multiagent Architectures: A Quantitative Study." *International Journal of Agent-Oriented Software Engineering* 2: 34-61.

[36] Lopes, C. V., and Bajracharya, S. K. "Assessing Aspect Modularizations Using Design Structure Matrix and Net Option Value." *Transactions on Aspect-Oriented Software Development I*. Berlin, Heidelberg: Springer-Verlag.

[37] Rashid, A., and Aksit, M., eds. 2006. "Transactions on Aspect-Oriented Software Development (TAOSD)." In *LNCS 3880*, 1-35.

[38] Shaw, M., and Garlan, D. 1996. "Software Architecture: Perspectives on an Emerging Discipline." Upper Saddle River, NJ: Prentice-Hall.

[39] Douence, R., Fradet, P., and Sudholt, M. 2004. "Trace-based Aspects. Aspect-Oriented Software Development." Addison Wesley, 201-17.

[40] Atomikos. 2015. Extreme Transactions Atomikos Transactions. [Online]. Available http://www.atomikos.com.