Unified Conceptual Model for Joinpoints in Distributed Transactions

Anas M. R. AlSobeh, Stephen W. Clyde
Computer Science Department
Utah State University
Logan, Utah, USA
aalsobeh@aggiemail.usu.edu, Stephen.Clyde@usu.edu

***Abstract*—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. Aspect orientation has the potential of reducing this kind of complexity; however, currently, aspect-oriented programming languages and frameworks only allow weaving of advice into contexts derived from traditional executable structures. This paper lays a foundation for weaving advice into distributed transactions, which are high-level runtime abstractions. To establish this foundation, we capture key transaction events and context information in a conceptual model, called *Unified Model for Joinpoints Distributed Transactions* (UMJDT). This model defines interesting joinpoints relative to transaction execution and context data for woven advice. A brief discussion of advice weaving and the potential for reducing complexity with transaction-specific aspects is provided, but the details of the actual weaving are left for another paper. Also, this paper suggest further research for studying the modularity and reuse achieved through the ability to weave crosscutting concern into transaction directly.**

Keywords-complexity; modularity; distributed transaction; joinpoint; operation; context; advice; aspect; crosscutting concerns.

I. INTRODUCTION

Frederick Brooks characterizes software complexity as either *essential* or *accidental*, where essential complexity stems from the very nature of the problem being solved by the software and accidental complexity comes from the way that the problem is being solved [1]. A *Distributed Transaction Processing System* (DTPS) may have essential complexity in the nature of the data, operations on the data, or the volume of data. However, issues such as logging, persistence, resource location, and even distribution itself are more likely to be sources of accidental complexity, because they are not usually inherent parts of the problem. When these issues are secondary to the primary purposes of a DTPS, it is common to find logic for them scattered throughout the software and tangled into core application logic. For example, concurrency-control operations, like locking and unlocking, may be spread throughout the system and be implemented with similar snippets of code.

Aspect Orientation (AO), an extension to *Object Orientation* (OO), can help manage both essential and accidental complexity by localizing and encapsulating crosscutting concerns in first-class software components, called *aspects* [2]. An aspect is very much like a class in OO 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*. However, existing AO Programming Languages (AOPLs) and frameworks only allow the weaving of advice into the execution of code-based contexts, such as methods, constructors, and exceptions. They do not directly allow behaviors to be woven into more abstract contexts, such as transactions.

One could argue that a good programmer can do the same thing in OO by defining classes for the crosscutting concerns and hard coding calls to methods of those classes in all the right places. However, the issue is not whether it can be done; rather, it is the difference in abstractions. AO 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 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* [3].

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, other concurrent transactions cannot see the effects of a transaction until it completes, 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 [3][5].

Distributed transactions are transactions, but their operations are executed 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 still observe the ACID properties.

Regardless of whether a distributed transaction is a flat sequence of operations or comprised of nested transactions, it is an ephemeral concept that spans multiple execution threads and operations using distributed resources. Therefore, from an execution perspective, it may seem non-contiguous and unevenly spread over time and space. A transaction's context is not tied to code constructs, like constructors and

 ISBN: 978-1-61208-367-4

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 its very difficult for AO developers to localize and encapsulate crosscutting concerns that apply to transactions as execution units.

This paper takes a preliminary step in enabling AO developers to treat transactions as first-class concepts into which compilers or frameworks can weave crossing concerns. 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.

Section II provides more detail about aspect-oriented programming concepts and background about common transaction concepts. Section III proposes possible joinpoints in the execution of distributed transactions and relevant the context information for each. Section IV presents a sample of transaction-related crosscutting concerns. Section V presents the UMJDT model and discusses two key areas, namely a transaction context and joinpoints. Although the technical details of advice weaving are beyond the scope of this paper, Section VI provides an outline of the process and highlights some of the key issues. Section VII summarizes the contributions on this paper and discusses next steps.

II. BACKGROUND

A. Overview of Aspect Orientation

As mentioned above, AO is an extension to OO that allows developers to extract and untangle secondary concerns from the primary features of an application. It is difficult to define what constitutes a secondary concern in general because it depends on the purpose of the software being built. However, secondary concerns often show up in less-than-expertly-designed OO software as similar snippets of code scattered across multiple modules or tangled into methods that primarily serve other purposes. A common example is tracing or logging in a data processing application, where the developers want a chronology of the execution for either system verification, audit-trail, or performance-monitoring measurement reasons. To do this, they might insert logic throughout the code that writes various messages or statistics to a file. Eventually, these log-writing code snippets become scattered across the software and tangled in otherwise cohesive methods.

An AOPL, like AspectJ [6], would allow a developer to remove all of the log-writing code from the main application and place that logic in an *aspect*, which is a class-like abstract data type. An aspect can include data members, methods, nested types and everything else a class can include. However, they can also include *advices* and *pointcuts*. An advice is like a method because it implements some specific behavior; however, it is not invoked like a method. Instead, the AOPL's compiler or runtime environment *weaves* the advice into the system so it is executed at specific places and time defined by *pointcuts*. A *pointcut* is a pattern that identifies a set of *joinpoints*, which are best characterized as intervals within program's execution flow. Examples of joinpoints in typical AOPL's include the execution of a method or the setting of a property. Consequently, their start and end points map to specific elements of the code, called *shadows*, which correspond to places where those intervals may start or end. The weaving of advice into the shadows is an automated process, and understanding it in depth is not necessary to appreciate the contributions of this paper. We refer readers interested in learning more about weaving of advice to the overview of AspectJ by Kiczales, et al. [6].

When advice executes, it can access context information about the joinpoint at which it was invoked. This context includes the location of the joinpoint (i.e., the shadow) and runtime information about the objects involved. Some on the context information is static and therefore can be computed during weaving; other context is dynamic and depends on the objects involved in the joinpoint.

B. Transaction Concepts

As mentioned, the objective of this paper is to lay the foundation for weaving crosscutting concerns into transactions in DTPS's. This requires identifying the logical places, i.e., joinpoints, in transaction execution where a developer might want to weave advice, as well as the kinds of information that should be available in joinpoint context.

There are many different DTPS's in use today and they vary in terms of features and implementations. However, they share commonalities in their underlying concepts of transaction distribution, management, execution, and concurrency control. It is on these basic concepts that we will focus our attention and lay a foundation for identifying transaction joinpoints and context.

As with transactions in centralized systems, a distributed transaction is a sequence of operations on shared resources that observe the ACID properties [7][8]. The difference is that the operations of a distributed transaction execute on more than one host machine, which opens up the possibility of subsequences of those operations executing concurrently, without shared memory to help with concurrency controls.

In general, a distributed transaction can be thought of as a tree of operations, instead of strict sequence. To visualize this, consider a simple example of a transaction-based manufacturing system that builds *Widgets* from *Goo* and *Gadgets* from *Widgets*. See Figure 1. The *Goo*, *Widget*, and *Gadgets* are all stored in "piles". The individual objects and the piles of objects are all shared resources. This system also includes processing components, i.e., shared resources, that handle the manufacturing. Specifically, there are *Builders* that create *Raw Widgets* from *Goo*, *Bakers* that turn *Raw Widgets* into *Rough Widgets* and *Polishers* that refine *Rough Widgets* into *Polished Widgets*. Finally, there are *Assemblers* that create *Gadgets* from *Widgets* and *Labelers* that tag the

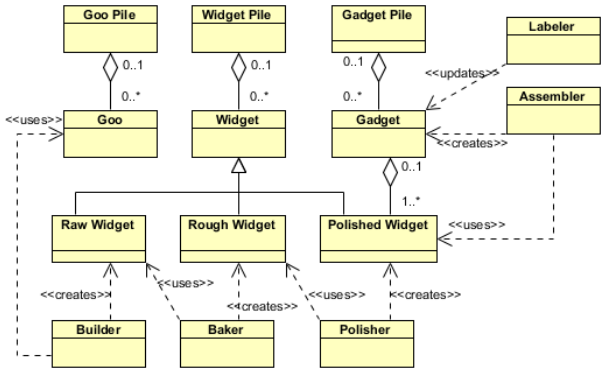

Figure 1 - Resources in a Widget and Gadget Manufacturing System.

a) Transaction T1
Op1.1: Get Goo from Goo Pile
Op1.2: Give Goo to a Builder and get back a Raw Widget
Op1.3: Give Raw Widget to a Baker and get a Rough Widget
Op1.4: Give Rough Widget to a Polisher and get a Polished Widget
Op1.5: Put Polish Widget in a Widget Pile

b) Transaction T2
Op2.1: Get Widget (W1) from Widget Pile 1
Op2.2: Get Widget (W2) from Widget Pile 2
Op2.3: Give W1 and W2 to Assembler and get a Gadget, G
Op2.4: Put Gadget G in a Gadget Pile
Op2.5: Have Labeler put a tag on G

Figure 2 - Two Sample Transactions for Constructing Widgets and Gadgets.

Gadgets with serial numbers. Figure 2 lists two simple transactions that represent a) the construction of a *Polished Widget* and b) the construction of a *Gadget* from two *Widgets.*

Now assume that piles of *Goo*, *Widgets*, and *Gadgets* are distributed across many locations (hosts) and that *Builders* are at the same location as *Goo Piles*; *Bakers* and *Polishers* are at the same location as *Widget Piles*; and *Assemblers* and *Labelers* are close to *Gadget Piles*, but not necessarily at the same location. With this distribution of resources, transaction *T2* could execute in a distributed manner by having *Op2.1* execute in a sub-transaction, *ST2.1*, *Op2.2* execute in another sub-transaction, *ST2.2*, both on the same host as the desired *Widget Pile*, and *Op2.3*-*Op2.5* in a sub-transaction, *ST2.3*, on the same host as the desired *Gadget Pile*. Figure 3 represents this distributed transaction as a simple tree with *T2* as the root and the operations as the leaves.

T1 and *T2* are just two concrete transactions, but this system could have hundreds of similar transactions running at the same time. As in all DTPS, each transaction receives a unique identity, i.e., *Transaction Identifier* (TID), when it starts. All references to a transaction will be via this identifier. Typically, in a DTPS, a *Transaction Manager* (TM), is responsible for assigning TID's and keeping track of parent/sub-transactions relationships.

Beside TID assignment, TM's are also typically responsible for starting transactions (and sub-transactions), and ending transactions by either committing or aborting the results. A TM may also oversee the execution of transaction

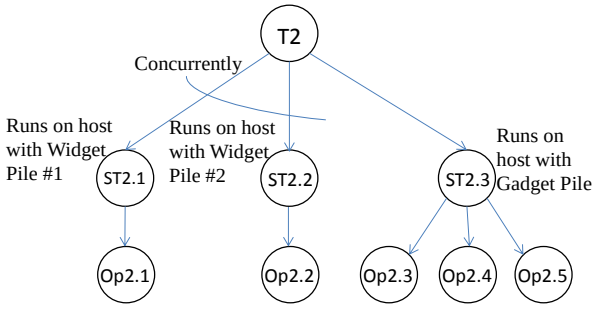

Figure 3 - Possible Distribution of Transaction T2.

operations on resources and any necessary concurrency controls, such as locking, for those resources. Some DTPS delegate these responsibilities to separate components such as *Resource Managers* and *Lock Managers*, but such architectural differences are not important here. For the purpose of exploring possible transaction-related joinpoints and context information, it is important to just recognize that operation execution and concurrency control take place with respect to individual resources.

Finally, a TM can also track information about its execution environment, including information about threads of execution, processes, host machines, secondary storage, and even network connections. It may do this for a variety of reasons, including performance management, audit trails, and recovery in case of failure.

A transaction is typically broken up into two basic phases: an execution phase and a commit phase [8]. The execution phase is considered tentative, because the changes are not made permanent until the commit phase. During the execution phase, the TM performs the operations in the body within its own context. Logically, the operations may result in the tentative changes to shared resources. In a commit phase, the TM will either finalize all of the tentative changes or abort the transaction.

Three common approaches to concurrency controls are *optimistic*, *timestamp-based*, and *pessimistic. Optimistic* approaches to concurrency control allow conflicts to occur during the tentative phases of concurrent transactions, then leave it up to the TM to detect conflicts and abort one or more transactions when they occur, using either *forward or backward validation* [9][10]. *Timestamp-based* approaches guarantee *serial equivalence* [11] by imposing an ordering on the execution of the operations in the tentative phase. *Pessimistic* approaches use locks to prevent conflicts from occurring in the tentative phase of execution. They do this by delaying operation execution or by trigging an abort (in the case of deadlock [12]). Locking schemes vary, but are all based on premise that a transaction must hold a particular kind of lock before performing an operation.

A common and simple locking scheme consists of two types of locks: one for read operations and one write for operations [12]. The pseudo-code in Figure 4 includes requests for the appropriate read and writes locks, following this simple scheme.

 ISBN: 978-1-61208-367-4

A transaction's context information includes those pieces of data and metadata that the transaction needs to be self-contained, guarantee the ACID properties, and support correct execution of both the tentative and commit phases of execution. Supporting correct execution of the commit phase means that the context needs to include sufficient information for the TM to decide whether the transaction conflicts with other concurrent transactions. However, the details of this context data depend heavily on the implementation of the DTPS, the types of concurrency control in use, and the commit algorithm. The only data that are common to virtually all DTPS are the TID and a reference (direct or indirect) to the responsible TM. Beyond these two items, a transaction's context may include many different kinds of implementation specific data, e.g., sets of tentative values, rollback logs, snapshots, lock information, timestamps, and other kinds of metadata. Therefore, any system that aims to support aspects for transaction must allow for context information to contain data that specific to a DTPS's implementation.

III. POTENTIAL JOINPOINTS AND THE SCOPE OF THE CONTEXT

From an advise-weaving perspective, joinpoints map to places where weaving takes place – hence the user of "point" in the name. However, from an execution perspective, a joinpoint represents a logical interval of time in a flow of execution. It has a beginning and an end, and advice can be woven into the flow of execution before, after, or around it. This section presents Figure 4 as a pseudo-code for an implementation of *T2* annotations that illustrate five new types of joinpoints for DTPS's: *outer transaction*, *inner transaction*, *resource locked*, *locking*, and *operation*. Each type of joinpoint is in a different color. This section also discusses interesting metadata that advice might want to use, and therefore should be part of joinpoint contexts.

An *Outer Transaction Joinpoint* represents an interval that spans the complete execution of a transaction, starting just before the tentative phase and ending after the completion on the commit phase. This kind of joinpoint would allow a programmer to introduce advice before, after or around an entire transaction. However, because it starts before the beginning of the tentative phase, any "before" advice would not have access to the target transaction's context information. However, it would have access to a parent transaction's context, which would be particularly important for advice before or around sub-transactions.

An *Inner Transaction Joinpoint* is similar to an *Outer Transaction Joinpoint*, except that it starts just after the tentative phase begins and ends just before the commit phase ends. Advice woven before this kind of joinpoint would have access to the target transaction's context.

Resource Locked Joinpoint represents an interval that spans the time when a lock is held, starting after acquiring of the lock and ending just before its release. Advice woven before, after or around this type of joinpoint would have

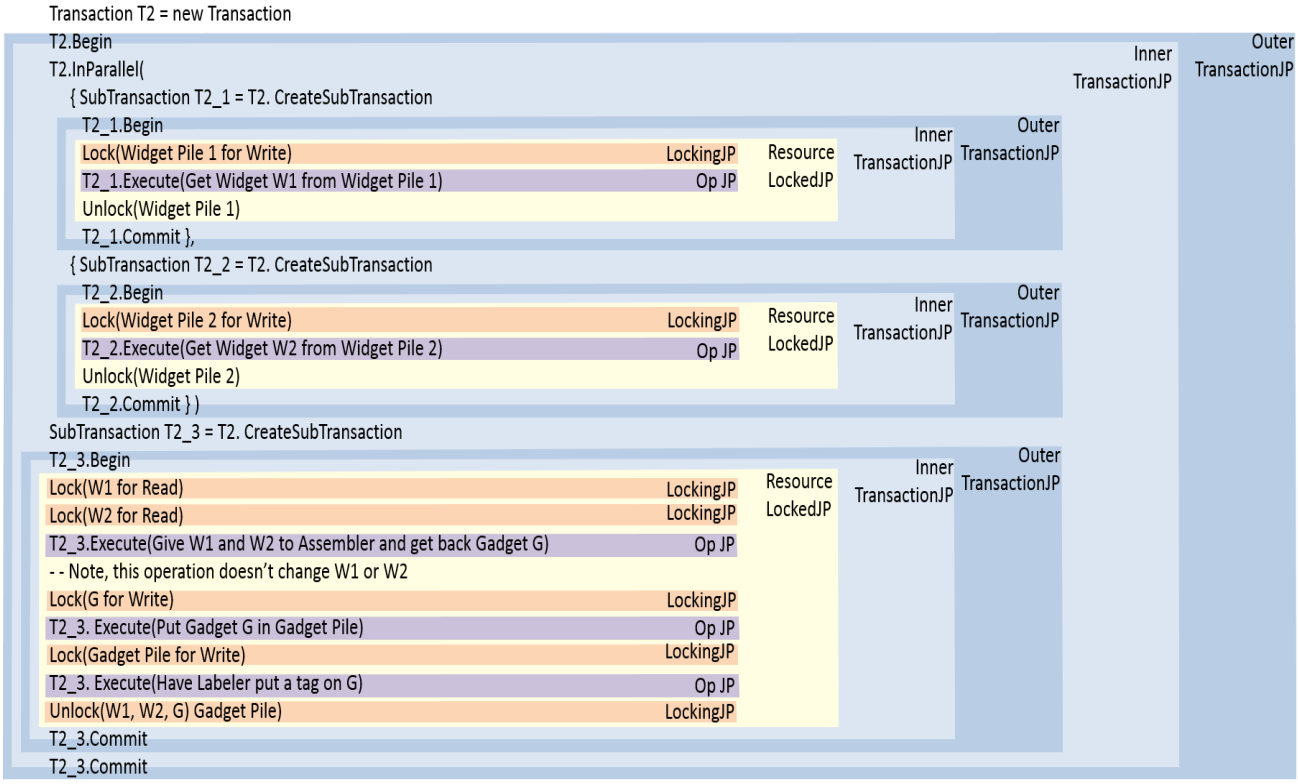

Figure 4 - Pseudo Code for Distributed Version of T2 and the Potential Transaction Joinpoints within the Scope of the T2's Context.

access to metadata about the lock, the associated resources and, of course, the transaction.

Locking Joinpoint represents an interval that spans a lock request. In other words, it begins as a request is made and ends when the request is granted or denied. Advice woven before, after, or around a *Locking Joinpoint* can access metadata about the type of lock being requested or the resource.

Operation Joinpoint is an interval that spans one operation in the execution of the tentative phase of a transaction. Such advice would have to access to metadata about the operation and the affected resources, as well as the transaction as large.

IV. SAMPLE CROSSCUTTING CONCERNS

The number and variety of crosscutting concerns in a DTPS are perhaps infinite. However, for illustrative purposes, we will consider just one here. Imagine that we would like to optimize the Gadget manufacturing system such that Widgets were created just in time, by making sure there are always some Widgets in a pile, but never an excess.

Such flow-control or timing issues could be considered a secondary crosscutting concern to the basic Gadget assembly problem. By talking with the domain experts, we would probably discover a couple of basic rules that govern when the Widget product needs to be speed up or slowed down. An OO programmer could embedded the logic for these rules into the implement of the *Builder*, *Baker*, *Polisher*, or some other set of components. With some skill, it is possible that the OO programmer might even be able to do this in a modular and reusable way.

With transaction aspects, an AOP programmer, however, would have a much similar option. Basically, the programmer would encapsulate the logic for speeding up or slowing down widget production into an aspect, maybe called something like *WidgetProductionSpeedControl*. This aspect would include advice that could be woven before (or around) any operation that accesses a widget pile. The advice's logic would speed up Widget product if the pile was getting too small or slow it down if the pile was getting too large. The aspect would also include a simple pointcut that defined a pattern for all relevant joinpoints. The original application code would not need to be aware of the new production-speed control logic. In fact, because of this obliviousness, it could be tested with or without the speed control functionality without any reprogramming of the system.

V. THE UNIFIED MODEL FOR JOINPOINTS IN DISTRIBUTED TRANSACTIONS

Figure 5 shows part of the UML model, called the *Unified Model for Joinpoints in Distributed Transactions* (UMJDT), which captures the key ideas for the new transaction joinpoints and related context information. The class labeled TransJP is a generalization of the joinpoints discussed in Section III. By definition, each is associated with a StartEvent, but may not have an EndEvent if the interval is still in process. Every TransJP can also reference a context

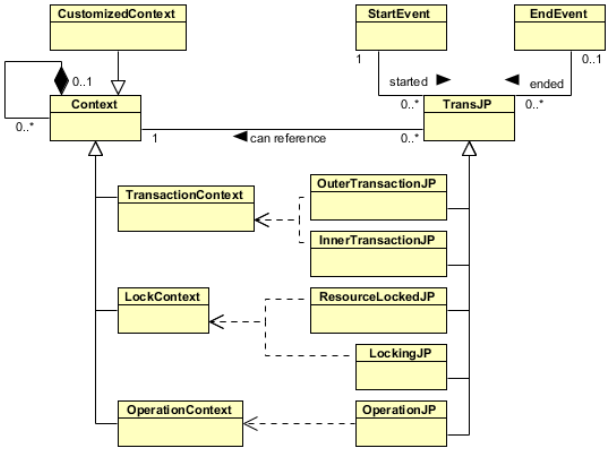

Figure 5 – Part of the Unified Model for Joinpoints in Distributed Transactions

that holds all the relevant statics and runtime information for the joinpoint. Aspect advice will use this context to access a wide variety of information such as operations in progress, resources, and current execution environments.

However, there are three special kinds of contexts, and the actually kind of context that a TransJP directly accesses depends on the TransJP specialization. For example, a LockingJP directly accesses a LockContext.

Contexts can be composited into a hierarchy of objects, as indicated by the recursive aggregation relationship connected to the *Context* class. Although Figure 5 does not show all the possibilities and constraints, a LockContext can be part of a TransactionContext, which could in turn be part of another TransactionContext (i.e., for a parent transaction.)

Contexts may also be extensible or customizable objects. In other words, the base system that makes transaction aspect possible, will provide classes for Context and its three immediate specializations. It also projects hooks for extending those classes, either through specialization, plugs-in, or even other kinds of aspects, so programmers can use context details that are specific to a particular DTPS or DTPS-based applications.

VI. ADVICE WEAVING

Kizcales, et al. introduced the idea of weaving logic for crosscutting concerns into core applications over 15 years ago [2]. Their work stems from even earlier research with inheritance, aggregation, and mix-ins [13]. Like all great ideas, the heart of the weaving solution is relatively straight forward – 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.

The challenge for transaction-related aspects is not so much the basic weaving process as it is pulling together all of the relevant data that needs to make up a transaction's

 ISBN: 978-1-61208-367-4

context. Remember, that in a DTPS, the execution of a single transaction is an abstraction that might span many different hosts and be interleaved with the execution of many other concurrent transactions.

So to solve this problem, we propose to build a runtime extension to AspectJ that tracks the start and end events of the TransJP's using low-level distributed aspects. We believe this to be feasible because it is similar to the technique used by CommJ to add communication-related aspects to AspectJ [14].

Although our approach will re-use many of the ideas first prototyped and refined in CommJ, our implementation for the weaving of transaction aspects will have to solve some additional problems not addressed by CommJ. Some of these problems include data-sharing optimizations, like the sharing of context information sharing across hosts only when necessary. Our future work will include research into both static and dynamic analysis techniques for solving these problems.

For the moment, solving the basic weaving and context management problems are sufficiently interesting and potentially beneficial to dominate our immediate attention.

VII. SUMMARY AND FUTURE WORK

This paper presented a foundation for extending AspectJ to support transaction aspects, using joinpoints and context information that is both interesting and relevant to DTPS's. In doing so, it paves the way for the weaving of crossing cutting concerns into high-level program abstractions that span multiple threads of execution and may be interleaved with concurrent execution of similar abstraction.

The main contribution of this paper is simply to identify the set of joinpoints and context information that make the most sense for DTPS's. We have captured this knowledge in a formal model called, *Unified Model for Joinpoints in Distributed Transactions* (UMJDT), as presented its essential parts here.

Our next steps are to a) complete the implementation of the an extension to AspectJ that performs the expected weaving and tracking of context information, and b) perform an preliminary experiment that we hope will provide evidence of improvement in modularization and reuse. To measure the modularity and reuse, we will define an extension to an existing quality model with following new factors: correctness, separation of concerns, understandability, obliviousness, throughput, transaction volume, transaction velocity, and transaction size. Each factor can be measured using metrics, such as diffusion of application, concern diffusion over operations, the number of inter-type declarations, the number of committed transactions, the number of aborted transactions, a rate of data flow during transaction executions, and the length of a transaction design and code, such as the lines of code, the number of operations, the number of components, i.e., classes and aspects, into the transaction, and the weighted operations per component. We also hope to create a toolkit consisting of reusable transaction aspects for common concerns, like performance measuring, logging, exception handling, audit trails, and tracing.

REFERENCES

[1] F. P. Jr. Brooks, "No silver bullet, essence and accidents of software engineering", Computer, vol. 20, no. 4, 1987, pp.10 - 19.

[2] G. Kiczales, et al. "Aspect-Oriented Programming," Proceedings of ECOOP '97, Springer Verlag, 1997, pp. 220–242.

[3] C. Clifton and G. T. Leavens, "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, December 2003.

[4] J. Gray and A. Reuter, Transaction Processing: Concepts and Techniques. Morgan Kaufmann, San Mateo, CA, 1993.

[5] G. Kohad, S. Gupta, T. Gangakhedkar, U. Ahirwar, and A. Kumar, "Concept and techniques of transaction processing of Distributed Database management system," International Journal of Computer Architecture and Mobility. (ISSN 2319-9229) Volume 1-Issue 8, June 2013.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and W. Griswold, "An Overview of AspectJ". 15th ECOOP 01, June 2001, pp. 327 – 357.

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

[8] J. Gray, "The Transaction Concept: Virtues and limitations", In Proceedings of the 7th International Conference on VLDB Systems (Cannes, France). ACM, New York, 1981, pp. 144-154.

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

[10] C. Mohan, B. Lindsay, and R. Obermarck, "Transaction Management in the R*· Distributed Database Management System," ACM Trans. on Database Systems, vol. 11, no. 4, December. 1986, pp. 378-396.

[11] G. Colouris, J. Dollimore, and T. Kindberg, "Distributed systems, concepts and design," Addison-Wesley. Fourth edition 2005. ISBN-10: 9780321263544, 2005.

[12] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," ACM Computing Surveys Vol. 13 No 2, June, 1981, pp. 185-221.

[13] A. Przybyłek, "Systems Evolution and Software Reuse in Object-Oriented Programming and Aspect-Oriented Programming," J. Bishop and A. Vallecillo (Eds.): TOOLS 2011, LNCS 6705, 2011, pp. 163–178.

[14] A. Raza and S. Clyde, "Weaving Crosscutting Concerns into Inter-process Communications (IPC) in AspectJ," ICSEA 2013. Nov. 2013, pp. 234-240. ISBN: 978-1-61208-304-9. Venice, Italy.

[15] A. Hastings, "Distributed Lock Management in a Transaction Processing Environment," In Proceedings of IEEE 9th Symposium on Reliable Distributed Systems, Oct. 1990, pp. 22-31.

[16] B. Gallina, N. Guelfi, and A. Romanovsky, "Coordinated Atomic Actions for dependable distributed systems: the current state in concepts, semantics and verification means," In Proc.

18th IEEE Int. Symposium on Software Reliability (Sweden). Nov. 2007, pp. 29-38.

[17] E. Bodden, "Closure joinpoints: Block Joinpoints without Surprises," In Proceedings of the 10th international conference on AOSD, New York, NY, USA, ACM, 2011, pp. 117-128.

[18] F. F. Rezende and T. Härder, "Concurrency Control in Nested Transactions with Enhanced Lock Modes for KBMSs," In: Proc. 6th DEXA, London, UK, Sept. 1995, pp. 604-613.

[19] H. Kung and J. Robinson, "On optimistic methods for concurrency control," ACM Transactions on Database Systems, Vol. 6, No 2, June 1981, pp. 213-226.

[20] I. Mejía, "Towards a Proper Aspect-oriented Model for Distributed Systems," AOSD '11. ACM New York, NY, USA, March. 2011, pp. 83-84.

[21] J. Eliot and B. Moss, "Nested transactions: and approach to reliable distributed computing Tech," Report MIT/LCS/TR-260, Massachusetts Institute of Technology, 1981.

[22] J. Kienzle, R. Jiménez-Peris, A. Romanovsky, and M. Patiño-Martinez, "Transaction Support for Ada," In Reliable Software Technologies - Ada-Europe, Springer Verlag, 2001, pp. 290 – 304.

[23] K. Donnelly and M. Fluet, "Transactional events," Journal of Functional Programming, v.18 n.5-6, September 2008, pp. 649-706. [doi>10.1017/S0956796808006916]

[24] M. Atif, "Analysis and verification of two-phase commit & three-phase commit protocols," In Proceedings of the 5th ICET, IEEE, Oct. 2009, pp. 326 -331.

[25] N. Dhamir, D. N. Mannai, and A. Elmagarmid, "Design and implementation of a distributed transaction processing system", COMPCON '88, IEEE, New York, Mar. 4, 1988, pp. 185-188.

[26] P. Ram and P. Drew, "Distributed transactions in practice," ACM SIGMOD Record, v.28 n.3, Sept. 1999, pp. 49-55. [doi>10.1145/333607.333613]

[27] R. Banks, P. Furniss, K. Heien, and H. R. Wiehle, "OSI Distributed Transaction Processing Commitment Optimizations," ACM SIGCOMM Comput Commun Rev , Vol. 28, No. 5. ACM, New York, Oct. 1998, pp. 61–75. ISSN: 0146-4833.

[28] R. J. Walker and G. C. Murphy, "Joinpoints as ordered events: towards applying implicit context to aspect-orientation," Workshop on Advanced Separation of Concerns at the 23nd ICSE, 2001.

[29] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," ACM Computing Surveys, v.15 n.4, December. 1983, pp. 287-317. [doi>10.1145/289.291].

 ISBN: 978-1-61208-367-4