Extending the Palladio Component Model to Analyze Data Contention for Modernizing Transactional Software Towards Service-Orientation

Philipp Merkle
Chair for Software Design and Quality
Karlsruhe Institute of Technology (KIT)
76131 Karlsruhe, Germany
merkle@kit.edu

Holger Knoche
Software Engineering Group
Kiel University
24118 Kiel, Germany
hkn@informatik.uni-kiel.de

ABSTRACT

The performance of data-intensive software, such as most enterprise software, is determined by two types of contention: The contention for physical resources, and the contention for data accessed within transactions. When modernizing existing software towards service-orientation, especially the latter type of contention may increase significantly due to access restructuring. To ensure acceptable performance of the modernized application, an in-advance analysis of the expected performance is required. However, most design-oriented performance models do not provide appropriate means for modeling data contention. In this paper, we propose an extension to the Palladio Component Model, called PCM.TX, which enables the analysis of performance effects due to data contention.

1. INTRODUCTION

Transactions are ubiquitous in enterprise software. Usually characterized by the well-known ACID (atomicity, consistency, isolation, and durability) properties [2], they allow automated failure recovery by the database and are a proven mechanism to hide concurrency from developers [1]. Transactions are an integral part of many frameworks and platforms for developing enterprise software. Often, the developer only needs to annotate parts which should be executed in a transaction (e.g., Enterprise Java Beans), or a transaction is active by default and the application only has to decide whether it has succeeded or not (e.g., CICS).

Transactions affect performance due to their concurrency-hiding property, which limits the achievable amount of concurrency in an application. Traditionally, databases employ locking-based mechanisms to detect and resolve conflicts between concurrent transactions. Every transaction acquires exclusive locks on data items it wishes to change, and releases these locks upon termination. Depending on the implementation and desired isolation level, shared read locks may be required as well. If a lock cannot be acquired, the transaction blocks until it becomes available. The transaction throughput therefore depends on the degree of lock contention, which in turn depends on the duration locks are held. To ensure sufficient throughput, many IT operators configure the databases to cancel transactions whose duration exceeds a given threshold.

Over the last years, service-oriented approaches have gained much attention in enterprise software development. The platform-neutral abstraction endorsed by these approaches allows companies to re-use their existing software assets, often so-called legacy systems, by providing their functionality as services. This process is commonly referred to as *wrapping* (e.g., [8]). The service abstraction also allows to transparently replace the underlying implementation and thus to incrementally modernize an existing system.

Unlike wrapping, which only adds a new interface layer, such a modernization entails substantial changes to existing implementations. The code needs to be *disentangled* [5], i.e., existing native accesses are eliminated and replaced by service invocations. Disentangling often requires changes to database accesses, as tables belonging to other services may no longer be accessed directly (e.g., using a join). Such changes may have substantial repercussions on performance. With respect to transactions, it is of particular importance that (i) a (potentially remote) service invocation in a running transaction may increase the transaction's duration significantly, and (ii) if an invoked service participates in a running transaction, the transaction may become distributed and require additional coordination overhead.

Especially the notion of microservices, which has recently gained popularity, emphasizes strict separation of service implementations, including their databases [6]. In such distributed environments, transactions reaching across multiple services can cause performance bottlenecks. Therefore, less rigorous transaction concepts have evolved, such as compensation or the Try-Cancel-Confirm (TCC) pattern [7].

When modernizing existing software, it is mandatory to keep it production-ready at all times. As the effects described above may endanger production-readiness due to increased response times or even timeouts, it is necessary to analyze in advance where and to what extent such effects may occur.

Unfortunately, most design-oriented performance models do not explicitly support the modeling of transactions. Synchronization primitives like semaphores, called *passive resources* in the Palladio Component Model (PCM), may be used to model transactions on a low level. The modeler, however, requires in-depth knowledge on locking schemes. The chosen locking scheme will be scattered all over the model making it cumbersome and error-prone to adjust the locking granularity, or the level of transaction isolation. We therefore propose an extension of the PCM, called PCM.TX, which allows to model database queries and transactions explicitly in order to simulate their expected runtime behavior. For service-oriented environments, we additionally propose means to facilitate modeling of compensation and TCC.

2. PCM.TX METAMODEL

The PCM.TX metamodel allows component developers to model database queries and transactional behavior. The newly introduced role of the database engineer is responsible for defining the database schema and for deploying database tables onto database servers. Queries may demand two types of resources, processing resources leading to hardware contention, and data resources leading to data contention. PCM.TX entirely reuses the PCM's capabilities to model hardware contention and focuses on providing modeling extensions for data contention. On this basis, an extended PCM solver could predict lock waiting times and aborts caused by timeouts.

Our approach targets relational databases due to their ubiquity in enterprise software, but is not limited to them. Document-oriented databases, for example, are organized as *collections* of *documents*. Collections correspond to tables, and documents are much like rows, even if they do not follow a rigid schema and tend to be more denormalized. PCM.TX extends the PCM in a non-intrusive manner using the *PCM profiles* [4] extension mechanism, in conjunction with subclassing.

2.1 Overview

PCM.TX conceptually distinguishes three types of components that participate in database accesses: (i) *Application components* contain the business logic without accessing the database by themselves. Instead, they call external services of (ii) *data access components* that map service calls to table accesses by means of queries. Queries indirectly refer to tables provided by (iii) *database components*. It is important to note that database components are not modeled explicitly but arise from defining and deploying tables onto database servers. While no deployment restrictions apply to the application component, the data access component must be deployed on a database server that provides its required tables. This ensures that processing as well as table demands of queries take effect on the database server. The application server and the database server may be the same machine, which is important for many legacy applications.

2.2 Elements of the PCM.TX Metamodel

This section describes the elements of the PCM.TX metamodel. Please note that Figures 1-3 depict metaclasses from the existing PCM in lighter colors compared to metaclasses introduced with PCM.TX. Existing associations and dependencies are marked by an arrow symbol.

2.2.1 Databases, Tables, and Entity Types

Fig. 1 illustrates how the database schema is modeled, and how tables are deployed on database servers. In PCM.TX, all data resides in *tables*. Each table belongs to exactly one *database* and is deployed on not more than one *database server*, a specialized *resource container* tagged with the corresponding stereotype. Tables contain a number of *rows*, which are the atomic units with respect to data contention. However, the rows themselves are not modeled. Databases have a default *isolation level* and may specify a *timeout* after which transactions abort. Tables and databases reside inside a *data repository*, which serves as a simple container without semantic relevance.

In order to keep the database schema specification separated from component specifications, data access components refer to *entity types* rather than tables. An entity type represents a meaningful group of data that is distinctively identifiable. The database engineer defines the schema by mapping each entity type to one or more tables. This way, changes to the database schema do not affect component specifications. It is assumed that each entity occupies exactly one row in each referred table, and different entities of the same type occupy different rows. Furthermore, all tables referred to by an entity type must reside on the same server.

Entity types are *resource types*, which allows them to be provided by database servers, and to be required by data access components by means of *resource interfaces* [3]. This ensures that context dependencies of components are made explicit as demanded by [9]. Consequently, data access components must be deployed on a database server that satisfies the component's entity dependencies.

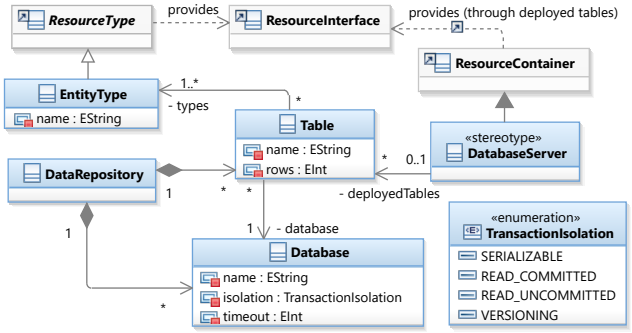

Figure 1: Schema definition and table deployment

2.2.2 Queries

Queries are intended to support modeling of data access components. As shown in Fig. 2, a database query is modeled as an *internal action* that demands entities in addition to processing resources. For this, we introduce the stereotype *entity access* that can be applied to *resource calls*. Multiple entity accesses can be added to the same internal action, allowing for representing queries that involve different entity types. Each entity access specifies the type and the number of entities the query wishes to access. The entity type is determined by selecting the resource interface provided by the requested entity type. Prior to that, the enclosing component needs to import the interface using a resource required role. To distinguish between read and write access, resource interfaces contain different signatures. The number

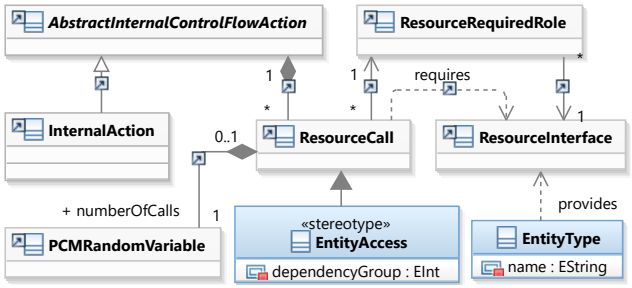

Figure 2: Modeling queries using resource calls

of accesses is a *PCM random variable* and may as such be a constant, a probability distribution function or a function of given parameters.

Entity accesses can be grouped to form *dependency groups*. Different accesses of the same dependency group are considered to be stochastically dependent, while accesses from different groups are considered stochastically independent. To illustrate this, consider two entity types *person* and *address*, both of which are mapped to the same table. A query that accesses, say, 50 persons and 50 addresses could mean that 50 persons along with their individual addresses are to be accessed, or it could mean that accessed addresses are unrelated to the accessed persons. To model the first case, the same dependency group would be assigned to both entity accesses, resulting in 50 rows accessed, while 100 rows would be accessed in the stochastically independent case.

2.2.3 Transaction Demarcation

Transactional RDSEFFs may be defined for application and data access components. PCM.TX supports both *declarative* (managed) and *imperative* (manual) transaction demarcation, as Fig. 3 shows. In both cases, the RDSEFF is annotated with the *transactional* stereotype, inspired by annotations from Java EE. For declarative transaction demarcation, the *scope* attribute defines if a service S2 called from service S1 participates in S1's transaction (*join*), or if S2 executes in its own transaction while suspending S1's transaction in the meanwhile (*new*). If S2 is not transactional, S1 will also be suspended until S2 returns. For imperative transaction demarcation, the scope can be set to *manual*. Only then *commit* and *abort actions* are permitted in the RDSEFF to control transaction boundaries. There is no corresponding *begin action* as a new transaction starts implicitly right after a commit, and with the start action.

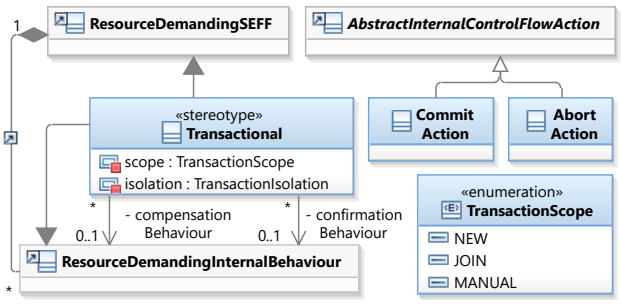

Figure 3: Transactions, compensation and TCC

2.2.4 Compensation and Try-Cancel-Confirm

Compensation and Try-Cancel-Confirm are alternative concepts for transactions in service-oriented contexts. Compensation implies that explicit compensation actions are taken by the client in case of transaction failure. With TCC, changes are explicitly tried (e.g., by creating a flight *reservation*) and confirmed in case of success (e.g., by turning the reservation into a booking). Explicit cancellation is optional in TCC. PCM.TX supports both concepts by allowing to specify confirmation and compensation behavior that is executed depending on the outcome of the transaction. Note that this behavior is part of the application component, and not the data access component.

3. CONCLUSION AND FUTURE WORK

Our approach PCM.TX allows to model, and soon to evaluate, design alternatives specific to transactional enterprise software. With the inclusion of compensation and TCC, we allow to combine traditional transactionality with service-oriented transactional concepts, thus supporting the migration of existing software towards service-orientation. We are already working on extending the Palladio simulation to cover modeling elements introduced with PCM.TX. In this process, we expect minor adjustments to the metamodel. Especially non-uniform data access distributions have been neglected for reasons of space. Also, sharding, i.e. horizontal partitioning of tables, is a candidate for future work.

4. REFERENCES

[1] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The Recovery Manager of the System R Database Manager. *ACM Comput. Surv.*, 13(2):223–242, June 1981.

[2] T. Haerder and A. Reuter. Principles of Transaction-oriented Database Recovery. *ACM Comput. Surv.*, 15(4):287–317, Dec. 1983.

[3] M. Hauck, M. Kuperberg, K. Krogmann, and R. Reussner. Modelling Layered Component Execution Environments for Performance Prediction. In *Proceedings of the 12th International Symposium on Component Based Software Engineering (CBSE 2009)*, number 5582 in LNCS, pages 191–208. Springer, 2009.

[4] M. E. Kramer, Z. Durdik, M. Hauck, J. Henss, M. Küster, P. Merkle, and A. Rentschler. Extending the Palladio Component Model using Profiles and Stereotypes. In *Palladio Days 2012 Proceedings*, Karlsruhe Reports in Informatics, pages 7–15, 2012.

[5] S. Murer, B. Bonati, and F. J. Furrer. *Managed Evolution: A Strategy for Very Large Information Systems*. Springer, Heidelberg, 2011.

[6] S. Newman. *Building Microservices*. O'Reilly, Sebastopol, CA, 2015.

[7] G. Pardon and C. Pautasso. Atomic Distributed Transactions: A RESTful Design. In *Proceedings of the 23rd International Conference on World Wide Web*, pages 943–948, 2014.

[8] H. M. Sneed. A pilot project for migrating COBOL code to web services. *International Journal on Software Tools for Technology Transfer*, 11(6):441–451, 2009.

[9] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, 2 edition, 2002.