Transactional CSP Processes

Gail Cassar and Patrick Abela

Ixaris Systems (Malta) Ltd
{gail.cassar|patrick.abela}@ixaris.com

Abstract. *Long-lived transactions (LLTs)* are transactions intended to be executed over an extended period of time ranging from seconds to days. A long-lived transaction is normally organized as a series of *activities,* with each activity being a discrete transactional unit of work that releases transactional locks upon its execution. The long-lived transaction commits if all its activities complete successfully. Unless an activity requires the result of a previously committed activity, there is no constraint which specifies that the various activities belonging to a long lived transaction should execute sequentially. In this paper we present a solution that combines long-lived transactions and CSP such that independent activities execute in parallel to achieve flexibility and better performance for long-lived transactions. We introduce two composition constructs SEQ_LLT and PAR_LLT. Very much as the occam CSP-based constructs, SEQ and PAR, allow processes to be executed sequentially or concurrently, the proposed SEQ_LLT and PAR_LLT constructs can be used to specify the sequential or concurrent execution of transactions. Transactional CSP Processes is a framework that makes use of these composition constructs, providing an API through which the application developer can define long-lived transactions. Concurrency and transaction handling are managed by the framework transparently from the application developer.

1 Introduction

An *atomic transaction* is a unit of interaction between two or more parties which must be either entirely committed (completed) or aborted (fails) as a unit. Transaction integrity is supported through the ACID properties, which are:

- *Atomicity*: ensures that all of the tasks in a transaction complete successfully and a transaction commits its changes. However, if any of the tasks fail, the transaction is aborted and all its effects are rolled back.
- *Consistency*: refers to having a legal state before a transaction begins and after it terminates. Integrity constraints of the database should be adhered to and a transaction is aborted if any of these constraints is aborted.
- *Isolation*: a transaction should be treated independently from any other transaction. Other transactions should not be aware of the intermediate states produced by the transaction before it commits.

– *Durability*: when a transaction commits, its effects are not lost.

A transactional platform makes use of locks on rows or tables to guarantee such ACID properties. The short-duration of such transactions allows other transactions competing for the same resources to be queued seamlessly until locks are released.
A long-lived transaction (LLT) also referred to as a long-running transaction, is a transaction of long duration, generally ranging from minutes to hours or even days. For this reason, it is impractical to use locks throughout the duration of the long-lived transaction to prevent concurrent access from other transactions (which would violate the ACID properties). To address this issue, transaction models for LLTs relax the ACID properties by organizing a long-lived transaction as a series of *activities*. Each activity is a discrete transactional unit of work which releases locks upon its execution. Activities are executed in sequence and can commit, rollback or suspend execution of the transaction. The long-lived transaction commits if all its activities complete successfully. If any of the activities fail, the long-lived transaction should roll back by undoing any work done by the already completed activities. This is normally achieved through compensating activities that essentially reverse changes which, under a normal setup, would have been handled implicitly by the underlying transactional model. As described in the JSR95 model [Com06], a transactional model proposed as part of the Java Community Process which can be used to model long-lived transactions,

> *"In the event of failures, to obtain reliable execution semantics for the entire long-lived transaction, compensation transactions are required in order to perform forward or backward recovery"* [Com06].

One limitation of traditional long-lived transactions is that activities are executed in sequence — the flow of execution continues upon the successful completion of an activity to the next. There might be cases when an activity may require a third party service and so it will be suspended, blocking other activities. This results in the long-lived transaction taking a long time to complete. There is also the possibility that after waiting for a long time to resume execution, the activity may fail causing any committed activities to undo their work. This leads us to the motivation of our research, which will be described in section 1.1.

1.1 Motivation

We shall now describe the motivation to develop a model which combines concurrent processes and long-lived transactions through an example. Consider a travel agent system which is used by clients to reserve resources such as a flight, a room and appropriate transport arrangements between the airport and the client's accommodation.
It is rather critical that the booking of such disparate resources is synchronized — it would be useless to reserve an accommodation unless we manage to reserve also a suitable flight. Traditional ACID transactions are not suitable to model such a transaction, as typically each of the flight reservation, accommodation

reservation and travel arrangement operations are done through different third parties and each takes a substantial amount of time. A long-lived transaction with a series of activities would be more suitable.

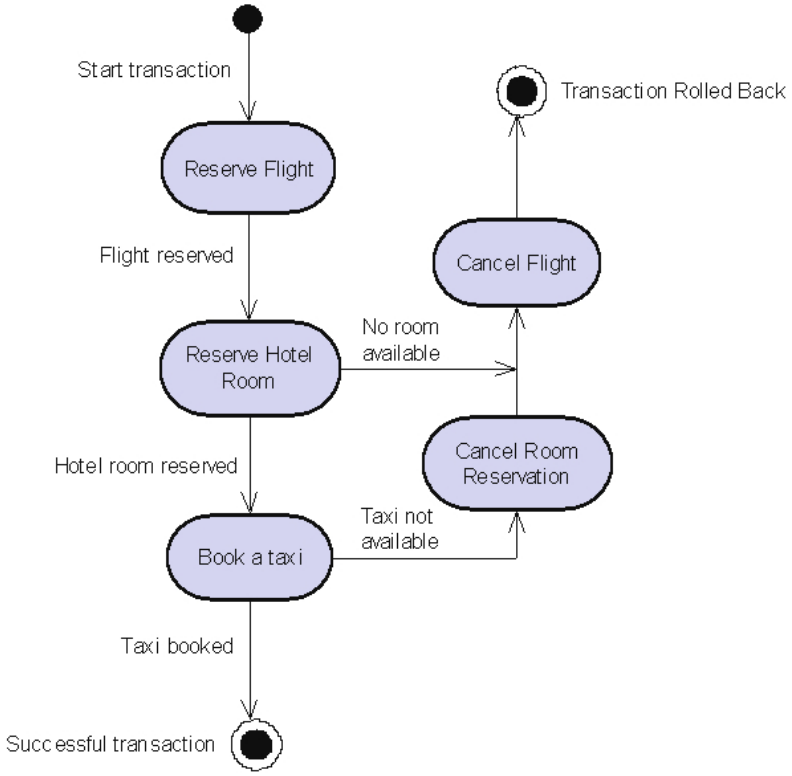

Fig. 1: Travel Agent LLT consisting of three activities to reserve a flight, a hotel room and a taxi, and a number of compensating activities to cancel each reservation in case of failure.

Typically we would model each of the various tasks as activities (refer to figure 1). If any of them fails, then any committed activities are compensated and the entire transaction is rolled back. A typical compensating action might involve canceling a committed room reservation as no suitable flight could be booked.

It is evident that there is no constraint which specifies that the various activities described in such scenario need to be executed sequentially. On the contrary, the booking is more likely to be more successful if we clear the reservations as quickly as possible. A solution would be to define a long-lived transaction that can have independent activities running in parallel without the need for them to wait for each other to start executing. Having concurrent activities would

eliminate the case where activities in suspended state will block the following activities.
Transactional CSP Processes is a framework implemented to achieve this objective. This framework extends on SmartPay LLT (refer to section 2) by introducing composition flow constructs similar to the sequential and parallel operators defined in the CSP calculus which allow an application developer to define the desired method of activity execution in a long-lived transaction.

1.2 Paper Overview

The paper is organised as follows: Section 2 provides a brief introduction to SmartPay LLT, an implementation by Ixaris Systems (Malta) Ltd. We will then present the Transactional CSP framework, which extends on SmartPay LLT in Section 3. This section presents details about the elements comprising the framework including activities, how the framework handles failure, as well as the composition constructs SEQ_LLT and PAR_LLT to be used in order to define the desired execution of activities in a long-lived transaction. In Section 4 we will highlight a possible extension to the framework which can be carried out as future work.

2 SmartPay LLT

In 2005, Ixaris Systems (Malta) Ltd developed a Java implementation loosely based on the JSR95 Model [Com06]. This implementation forms part of a generic SmartPay platform implemented by the same company.
The motivation for such a model was brought about by the inadequacy of traditional ACID transactions to address the company's specific circumstances. Generally financial transactions involve a mix of local database updates (fully within a transactional context) as well as external communication with third parties. The communication with such third parties cannot be done within a normal transactional context; if the third party communication falls through, it takes a period of time for the connection to timeout; during such a period all local resources participating in the transaction are locked. Long-lived transactions allow for the separation of remote interactions and local transactional updates.
The SmartPay LLT implementation extends the long-lived transaction model proposed in the JSR95 with the possibility of suspending execution between one activity and another. There exist situations when one needs to consult with a remote system before continuing with the transaction (for example, if we are not sure whether we have acquired funds from a client, then we need to suspend the transaction until we check with the remote system before depositing funds in the user's local account).
SmartPay LLT is implemented on standard Java technologies namely Java Transation API (JTA) and Enterprise Java Beans (EJBs). JTA provide the Java Platform with standards-based closed, top-level transaction support. EJBs provide a persistence model for persisting intermediate transactional state. SmartPay

LLT 1.0 has been deployed on a live system setup (on a JBoss application server with a MySQL backend) for the past two years, processing thousands of financial transactions.

3 Transactional CSP Processes Framework

Transactional CSP Processes is a framework which allows the application developer to define the desired method of activity execution (sequential, concurrent or a combination of both) in a long-lived transaction. This framework extends on SmartPay LLT by introducing composition flow constructs similar to the sequential and parallel operators defined in the CSP calculus. The model also allows for the suspension and resuming of activities and addresses failure of activities in terms of compensating activities. The application developer simply determines the activities to be performed and specifies their method of execution by using the appropriate composition constructs. Concurrency and transactional issues are managed by the framework implementation, transparently from the application developer.
A long-lived transaction in Transactional CSP Processes framework is defined in terms of activities and their composition structures, using the proposed sequential and parallel composition flow constructs SEQ_LLT and PAR_LLT. In the following sections we will present the various elements that comprise the Transactional CSP framework.

3.1 Activities

Similar to the SmartPay LLT, the application developer must implement activities to define the units of work of a Transactional CSP LLT. The method of execution for each activity is defined by adding the activity to the desired composition flow construct SEQ_LLT or PAR_LLT. In the Transactional CSP Processes framework, the long-lived transaction is modeled using a tree structure in which a branch determines the concurrent execution of the elements belonging to it. Activities are to be added as child elements to the required composition structures:

- SEQ_LLT for sequential composition
- PAR_LLT for parallel composition

3.2 Compensating Activities

This framework adopts backward recovery so when an activity fails, all previously committed activities are compensated through their corresponding compensating activities. A compensating activity essentially reverses changes which, under a traditional atomic setup, would have been handled implicitly by the underlying database model. The application developer specifies the course of action to be taken for each committed activity to undo its changes in a corresponding compensating activity.

It is our understanding that all activities related via SEQ_LLT and PAR_LLT constructs belong to the same transaction. The failure of any activity brings about the compensation of all committed activities participating in the same LLT. Compensating activities will be executed in the same order and under the same constraints as the activities being compensated. When compensating activities in a SEQ_LLT, their corresponding compensating activities are executed sequentially but in reverse order, while those activities in a PAR_LLT are compensated concurrently.

3.3 Composition Flow Constructs: SEQ_LLT and PAR_LLT

Very much as the occam CSP-based constructs SEQ and PAR allow processes to be executed sequentially or concurrently, the proposed SEQ_LLT and PAR_LLT constructs can be used to specify the sequential or concurrent execution of activities in a long-lived transaction.
Two activities that are coordinated with the SEQ_LLT construct (Figure 2) are evaluated in such a way that the second activity is executed only after the first activity commits. This corresponds to the SEQ construct which, from a concurrency perspective, executes in such a way that the second process starts its execution after the first process is complete. Therefore, SEQ_LLT requires a single thread of execution for its elements to execute in sequence.

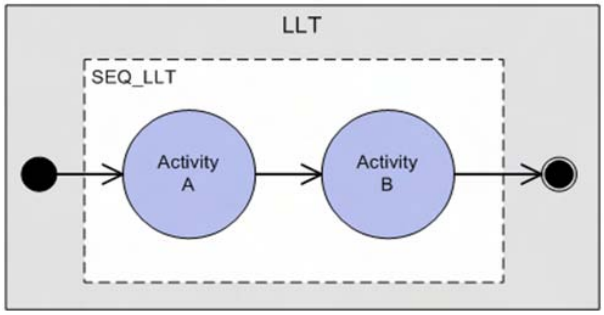

Fig. 2: SEQ_LLT construct is used to specify the sequential execution of activities in a long-lived transaction, with each activity executing one after the other on the same thread of execution.

Similar to occam's PAR construct, the PAR_LLT construct (Figure 3) specifies that activities can start their execution, independently from whether any other activities have committed their transaction or not. PAR_LLT will spawn a thread for each of its elements so that they will execute in parallel. PAR_LLT will then wait for all child threads to return a result to their original parent thread. In other words, PAR_LLT will wait for all child threads to join back to their original parent thread.

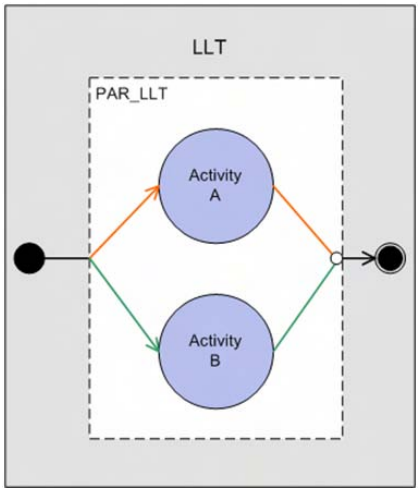

Fig. 3: PAR_LLT construct is used to specify the concurrent execution of activities in a long-lived transaction, with each activity being executed on a separate thread.

SEQ_LLT and PAR_LLT can be combined in very much the same way that SEQ and PAR in occam can. For example, if two activities B and C can run in parallel but require activity A to successfully commit first, the setup of SEQ_LLT and PAR_LLT as shown in Figure 4 is to be used.

In the case where a PAR_LLT construct is to be followed by a SEQ_LLT construct sequentially, like that depicted in Figure 5, the nested SEQ_LLT activities start execution after all activities in the PAR_LLT have committed. PAR_LLT will first spawn threads for each of its activities. PAR_LLT will wait for all its threads to join back to their original parent thread before the enclosing SEQ_LLT can proceed to execute the following element in sequence.

3.4 Suspending and Resuming Activities

An activity can commit its updates, rollback any updates or suspend execution such that it is resumed later on. Any updates done up to that point by the activity can be committed or rolled back, as specified by the activity.

An activity which suspends execution in a SEQ_LLT construct, indirectly delays the execution of any subsequent SEQ_LLT activities. Such activities cannot start their execution until the activity commits.

On the other hand, an activity which suspends execution in a PAR_LLT constructs, does not have any effect on other activities executing in the same PAR_LLT construct. Any activities which are synchronized to start their execution after the PAR_LLT activities commit will wait until the suspended activity is resumed and completed.

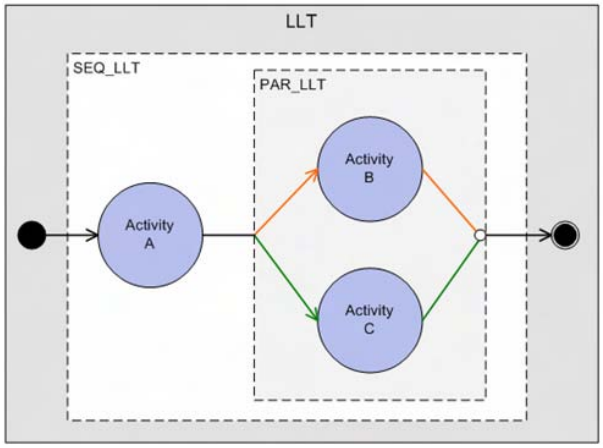

Fig. 4: A long-lived transaction made up of a nested composition of constructs in which Activities B and C first wait for Activity A to commit successfully, and then they are executed in parallel.

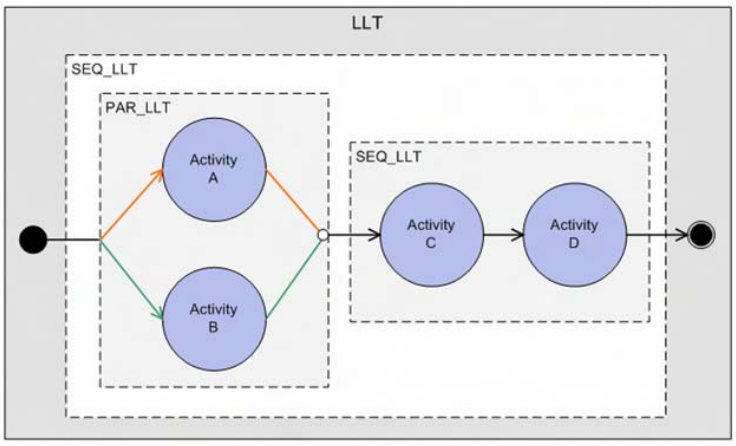

Fig. 5: Another example of a long-lived transaction made up of a nested composition of constructs, with a number of activities (C and D) to be executed in sequence, after a number of activities (A and B) have been executed successfully in parallel.

4 Future Work

Transactional CSP Processes framework could be extended to support communication between concurrent activities, using the same synchronization mechanisms provided by CSP.
An activity which waits on a channel for communication with another concurrent activity would be automatically suspended and its transactional locks released. Subsequently, it is resumed after it synchronizes. Effectively a received message on a channel causes the activity to resume its execution and to restart the transaction.

5 Conclusion

One can conclude that through the implementation of the Transactional CSP Processes framework, we have achieved the main objective of providing a solution that allows a long-lived transaction to have independent activities running concurrently. Support for concurrent activities provides more flexibility and better performance for long-lived transactions since activities running in parallel do not affect each other, thus avoiding scenarios where suspended activities cause a long-lived transaction to take a considerable amount of time to complete. The composition constructs SEQ_LLT and PAR_LLT, introduced and implemented by the Transactional CSP Processes framework, allow an application developer to define the desired method of execution for the activities in a long-lived transaction. Synchronization and transactional issues are managed by the framework transparently from the application developer, allowing more dedicated time towards the business logic of the transactional application.

References

[BHF05] M. Butler, T. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In *Proceedings of 25 Years of CSP*, 2005.

[Cas07] G. Cassar. Transactional CSP Processes. Department of Computer Science & AI, University Of Malta, 2007.

[Com06] Java Community. JSR 95: J2EE Activity Service for Extended Transactions. 2006.

[GMS87] H. Garcia-Molina and K. Salem. SAGAS. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, 1987.

[GR03] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 2003.

[Hoa87] C. A. R. Hoare. Communicating Sequential Processes. In *Proceedings of the ACM Programming Techniques*, 1987.

[LMP04] M. Little, J. Maron, and G. Pavlik. *Java Transaction Processing Design and Implementation*. Prentice Hall, 2004.

[Ros89] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1989.