Enforcing Commutativity Using Operational Transformations

Abdessamad Imine¹, Hanifa Boucheneb², and Michaël Rusinowitch³

1 Nancy-Université & INRIA Nancy Grand-Est, France
imine@loria.fr
2 Laboratoire VeriForm, École Polytechnique de Montréal, Canada
hanifa.boucheneb@polymtl.ca
3 INRIA Nancy Grand-Est, France
rusi@loria.fr

Abstract. Commutativity of operations is the better way to provide a high degree of concurrency on shared data types. In this short paper, we present a technique to increase concurrency using operational transformations. This technique enforces commutativity even though the operations do not naturally commute. We report our experience on (i) automatically verifying the correctness of this transformation-based commutativity and (ii) developing new applications based on operational transformations.

1 Introduction

When a shared object is modified by simultaneous operations, it is desirable to obtain as high a degree of concurrency as possible. Abstract Data Type (ADT) specifications contain information that can be used to increase concurrency. Several schemes based on the commutativity of operations have been proposed to provide more concurrency. If two operations commute then they can be executed in parallel. What happens if two operations do not commute? Let consider the following example where two threads T1 and T2 try to simultaneously modify a shared text containing initially the word “efecte”:

[T1: Insert(2,f)] and [T2: Delete(6)]. Thread T1 tries to insert the character “f” at position 2. Concurrently, thread T2 tries to remove the character “e” and position 6. It is clear that T1 and T2 are conflicting because their operations do not commute. Indeed, T1 followed by T2 produces the word “effece” while T2 followed by T1 results in the word “effect”. To ensure consistency, we can enforce a global serialization order of all operations by performing some roll-back or undo/redo operations. However, some global orders may be inconsistent with the original intention of some operations. Consider the order T1;T2: it is clear that the intention of Delete(6) (i.e., removing “e”) is not respected as another character is deleted.

In this paper, we sketch another solution that may enforce to some extent the commutativity between conflicting operations without using roll-back but by using transformations that must have been planned in advance. This solution, called Operational Transformation (OT), is originated in the context of collaborative editing systems [4,12]. The OT approach consists of application-dependent transformation algorithm IT(op1, op2) to compute a new variant of operation op1 that will be executed after operation op2. Thus, for every possible pair of concurrent operations, the application programmer has to define in advance how to merge these operations regardless of execution order. In our previous example, if we have planned in advance that IT(Delete(6), Insert(2,f))=Delete(7) and IT(Insert(2,f), Delete(6))=Insert(2,f) then both orders T1;T2 and T2;T1 give the same word “effect”. Intuitively we can write the transformation IT for the couple Insert(p1,c1) and Delete(p2) as follows:

IT(Insert(p1,c1),Delete(p2)) = if (p1 <= p2) return Insert(p1,c1) else return Ins(p1-1,c1);

The advantages of OT approach are: (i) it enables unconstrained concurrency, i.e., it requires no global order on concurrent operations; (ii) it transforms operations to run in any order even when they do not naturally commute; (iii) it produces a convergence state that precisely preserves the intentions of the original operations. OT has been used in many collaborative environments. More recently, it is used in Google Wave (a new google platform⁴).

The rest of the paper is organized as follows. Section 2 presents the ingredients of OT approach. In Section 3, we present our research works on how to verify formally the correctness of OT algorithms. Section 3 illustrates our achievements in this area.

2 Operational Transformation Approach

OT is an optimistic replication technique which allows many users (or sites) to concurrently update the shared object and next to synchronize their divergent replicas in order to obtain the same data. The updates of each site are executed on the local replica immediately without being blocked or delayed, and then are propagated to other sites to be executed again. Accordingly, every update is processed in four steps: (i) generation on one site; (ii) broadcast to other sites; (iii) reception on one site; (iv) execution on one site.

⁴ http://www.waveprotocol.org/protocol
It is very important to track the dependency between operations when they are generated by different users. So, an operation \( o_2 \) generated at site \( j \) causally depends on \( o_1 \) generated at site \( i \) (denoted \( o_1 \rightarrow o_2 \)) iff: (i) \( i = j \) and \( o_1 \) was generated before \( o_2 \); or, (ii) \( i \neq j \) and the execution of \( o_1 \) at site \( j \) has happened before the generation of \( o_2 \). Two operations \( o_1 \) and \( o_2 \) are said to be concurrent iff neither \( o_1 \rightarrow o_2 \) nor \( o_2 \rightarrow o_1 \). As a long-established convention in OT-based collaborative editors \([4, 12]\), the timestamp vectors are used to determine the causality and concurrency relations between operations.

Using the OT approach, each site is equipped by two main components \([4]\): the integration component and the transformation component. The integration component determines how an operation is transformed against a given operation sequence (i.e., the log buffer). It is also responsible for receiving, broadcasting and executing operations. It is rather independent of the type of the shared object. The transformation component is a set of IT algorithms which is responsible for merging two concurrent operations defined on the same state. Every IT algorithm is specific to the semantics of a given shared object.

It has been proved that any integration component can achieve convergence in the presence of arbitrary transformation paths if its IT algorithm satisfies two properties \( TP1 \) and \( TP2 \) \([11]\). For all \( op_1, op_1 \) and \( op_2 \) pairwise concurrent operations with \( op'_1 = IT(op_1, op_2) \) and \( op'_2 = IT(op_2, op_1) \), properties \( TP1 \) and \( TP2 \) are as follows:

\[
\text{TP1: } [op_1; op'_2] \equiv [op_2; op'_1]; \quad \text{and} \quad \text{TP2: } IT(IT(op, op_1), op_2) = IT(IT(op, op_2), op_1).
\]

Property \( TP1 \) defines a state identity and ensures that if \( op_1 \) and \( op_2 \) are concurrent, the effect of executing \( op_1 \) before \( op_2 \) is the same as executing \( op_2 \) before \( op_1 \). This property is necessary but not sufficient when the number of concurrent operations is greater than two. As for \( TP2 \), it ensures that transforming \( op \) along equivalent and different operation sequences will give the same operation. Properties \( TP1 \) and \( TP2 \) are sufficient to ensure the convergence for any number of concurrent operations which can be executed in arbitrary order \([11]\). Accordingly, by these properties, it is not necessary to enforce a global total order between concurrent operations because data divergence can always be repaired by operational transformation.

The integration component allows log of executed operations to be built on every site, provided that the causality relation is preserved (i.e., if \( o_1 \rightarrow o_2 \) then \( o_1 \) should be executed before \( o_2 \) at all sites). When all sites have performed the same set operations, log sites are not necessarily identical because the concurrent operations may be executed in different orders. Nevertheless, these logs must be equivalent in the sense that they must lead to the same final state. This equivalence is ensured iff the used IT algorithm satisfies properties \( TP1 \) and \( TP2 \).

### 3 Verification Methods

Finding an IT algorithm that satisfies \( TP1 \) and \( TP2 \) is considered as a hard task, because this proof is often unmanageably complicated. Consequently, to develop the OT approach and to safely use it in other concurrency-based systems with simple or complex shared objects, proving convergence of OT algorithms must be assisted by automatic tools. To the best of our knowledge, we are the first to propose verification methods for automatically verifying the correctness of OT algorithms. In this section, we briefly present our verification methods based on deductive and model-checking techniques.

**Verification based on Deductive Techniques.** In \([9]\), we proposed the modelling and deductive verification of OT algorithms with algebraic specifications. We modelled a shared object as an Abstract Data Type (ADT). Using observational semantics, we treated this ADT as a black box in order to abstract away from its internal state structure. Indeed, only interactions between a shared object and a user are considered. Operations for modifying the states are called methods and operations for observing the states are called attributes. We only access to the current state by observing its predecessor states modified by methods through attributes. We implemented our approach in a tool which enables a developer to define all ADT operations (methods and attributes) and the associated OT algorithm. From this description, our tool generates an algebraic specification described in terms of conditional equations. To verify properties \( TP1 \) and \( TP2 \), we used a first-order implicit induction prover, which is suitable for reasoning about conditional theories. Using our theorem-proving approach we obtained unexpected results. Indeed, we have detected bugs in several OT-based distributed groupware systems designed by specialists from the domain. Moreover, our verification method has been intensively applied to the design and debugging of a file synchronizer \([10]\) distributed with the industrial collaborative development environment, LibreSource Community\(^5\), proposed by Artenum Company.

When we consider a complex shared object (such as a filesystem or an XML document) the formal design of its OT algorithm becomes very tedious because of the large number of operations and synchronization situations to be considered if we start from scratch. In \([5, 7]\), we proposed a compositional method for specifying and verifying

\(^5\) http://dev.libresource.org/
complex ADTs. The most important feature of our method is that designing an OT algorithm for the composed object can be done by reusing the OT algorithms of component objects. By using our method, we can start from correct small objects (i.e. they satisfy properties TP1 and TP2) which are relatively easy to handle and incrementally combine them to build more complex objects that are also correct.

**Verification based on Model-Checking Techniques.** The verification method presented in [9] has some shortcomings: (i) it does not guarantee that the violation of property TP1 or TP2 is really feasible; (ii) there is no guidance to understand the counterexamples (when the properties are not verified); (iii) it requires some interaction (by injecting new lemmas) to complete the verification. To overcome these shortcomings, we proposed in [1] a model-checking technique to verify OT algorithms. Instead to verify properties TP1 and TP2, this technique is based on the generation of the effective traces of the system. So, it allows to get a complete and informative scenario when a bug (a divergence of two copies of the shared object) is detected. However, it needs to fix the shared object, the number of sites, the number of operations, the domains of parameters of operations and to execute explicitly the updates.

In [2], we proposed a symbolic model-checking technique to verify whether an OT algorithm satisfies properties TP1 and TP2. The verification of these properties is performed automatically without carrying out different copies of the shared object and executing explicitly the updates. So, there is no need to fix the state of the shared object. Thus, unlike [1], our symbolic model-checking technique enabled us to get more abstraction and to build symbolic counterexamples. Moreover, for fixed numbers of sites and operations, it allows to prove whether or not an OT algorithm satisfies properties TP1 and TP2.

**How to use these verification methods?** As they are the basis cases of the convergence property, TP1 and TP2 are sufficient to ensure the data convergence for any number of concurrent operations which can be performed in any order. Thus, the verification based on deductive techniques (e.g. using a theorem prover) remains better for proving that some OT algorithm satisfies TP1 and TP2. But it is partially automatable and, in the most cases, less informative when divergence bugs are detected. A model-checking based approach is fully automatable for debugging OT algorithms by finding divergence scenarios. Nevertheless, it is more limited as the convergence property can be exhaustively evaluated on only a specific finite state space.

4 Some Achievements

Our verification methods [1, 2, 5, 7, 9] are an important step towards facilitating the development of correct transformation algorithms. So, we think that our approach is very valuable because: (i) it allows OT developers to deal with the transformation for other ADTs; (ii) it can help significantly to increase confidence in a transformation algorithm; (iii) having a theorem prover or a model-checker ensures that all cases are considered and quickly produces counterexample scenarios. Interestingly, our verification methods help us to successfully perform the following achievements:

**Scalable and Decentralized OT-based Environments.** In [6], we proposed a new framework for managing collaborative editing work in real-time context. Using OT approach, it provides a simultaneous access to shared documents and an automatic reconciliation of divergent copies. A minimal causality between operations is given by means of a dependency relation based on semantics of the shared document. Unlike timestamp vectors [4, 12], our causality relation is not tributary to the number of users and it allows to support dynamic groups. Thus, our framework may be deployed in P2P networks. A prototype based on our OT framework has been implemented in Java. It supports the collaborative editing of wiki pages and it is deployed on P2P JXTA platform. Moreover, experimental results shown that our OT framework is fast and can be used safely even with large user groups.

**Optimistic Access Control.** Controlling access in collaborative editing systems is still a challenging problem, as they need dynamic access changes and low latency access to shared documents. In [3, 8], we proposed a flexible and generic access control model based on replicating the shared document and its authorization policy at the local memory of each user. To deal with latency and dynamic access changes, we used an optimistic access control technique where enforcement of authorizations is retroactive. We shown that naive coordination between updates of both copies can create security holes on the shared document by permitting illegal modification, or rejecting legal modification. networks. To validate our model, we implemented secured P2P editing tool and secured P2P shared calendar both deployed on mobile devices (such as IPhone and BlackBerry).

References