Operational Transformation and its Relevance in Games

Felix Pauck
6574788
fpauck@mail.upb.de

Patrick Steffens
7000302
psteffen@mail.upb.de

Abstract

Operational transformation is a mature technique for maintaining consistency in collaborative systems, also known as groupware. By presenting a case study, this paper illustrates that many games can be played collaboratively by using operational transformation. Based on a well-known consistency model described in this paper, it is evaluated which properties an operational transformation algorithm must have to guarantee in order to make a game collaboratively playable. Eventually, the findings of this paper lead to a definition of a class of games that can be realized with operational transformation, namely collaborative games.

1 Introduction

Collaborative software or groupware can be used to allow multiple users to work simultaneously on the same resource. Editors is one of the most common applications of collaborative software. Today there exist different versions of such editors for different resources. Google Docs[1], for example, contains editors for text documents, spreadsheets and presentation slides. Originally defined in a paper from 1989[1], Operational Transformation (OT) is still a state-of-the-art approach to realize such editors. As the name suggests, operational transformation is a technique to transform operations executed by different users on the same resource. The idea is to use such transformations to keep the resource consistent among all users. OT's benefit compared to other approaches is that the effect of an operation executed locally is immediately available without the system having to wait for a server processing the operation. Thereby, users of a collaborative system using OT, neither rely on a fast network connection nor do they have to wait for other users.

In the aforementioned paper about operational transformation [1], the authors mention games as an intuitive example for groupware: "An example that many find easy to relate to is the multi-player game". Furthermore, that the interest of playing a game collaboratively exists is demonstrated by the success of *Twitch plays Pokémon*[2], which is a social experiment started in 2014 and played by roughly a million players in total during the first two weeks. It is hosted on a streaming platform and allows an arbitrary number of users to play different editions of Pokémon games collaboratively just by sending commands via the chat of the streaming platform. Two different modes are supported: anarchy and democracy. In anarchy mode every command is executed immediately, whereas in democracy mode a command is only executed if there are enough players who invoke the same command within a certain period. With these very basic approaches of handling collaboration, the first Pokémon game could be finished within 16 days. Since a single player can provably finish the game much faster[3], it is obvious that there is no advantage in playing the game with such many players considering the time required to finish it.

[1]`https://docs.google.com`

[2]`https://www.twitch.tv/twitchplayspokemon`

[3]According to `http://wiki.pokemonspeedruns.com/index.php/World_Records`, the current world record for finishing Pokémon Red is 1:49h.

Nevertheless, until today it is neither known if games can be played collaboratively, using OT nor if it would be benefitial in terms of finishing a game more efficiently. In this paper, we show that it is possible and argue why it can be benefitial in specific games (see Section 5). To do this, we develop a simplified version of the well known single player game Solitaire which is described in Section 2. In Section 3, different consistency models are introduced which must be implemented by OT algorithms as described in Section 4 in order to enable playing this game collaboratively.

2 Simplified Solitaire

Solitaire is a famous card game that has been part of Microsoft Windows for decades now. In this paper a simplified version of this game is used as a running example. Furthermore, we explain how OT can be used to play this simplified version of Solitaire in a collaborative way.

The original Solitaire game is played with 52 cards, including 13 cards of each suit: ♣ Clubs, ♢ Diamonds, ♠ Spades, ♡ Hearts. The goal is to sort all cards to four cardstacks, one for each suit. The designated order is: A (Ace) $<$ 2 $<$ 3 $<$ 4 $<$ 5 $<$ 6 $<$ 7 $<$ 8 $<$ 9 $<$ 10 $<$ J (Jack) $<$ Q (Queen) $<$ K (King). To do so, 7 open stacks can be used to shift cards around and one stack can be browsed. We assume that the rules[4] are widely known and do not explain them in detail in this paper. Doubtlessly, in order to understand the collaborative part of the simplified version it is not required to know the rules of the game.

This simplified version is used because it allows us to focus on the interesting collaborative part instead of explaining details and special cases. First of all, in this version of Solitaire the number of suits and cards is reduced. Only the 13 cards of Clubs (♣) and Hearts (♡) are considered. Accordingly, the number of open stacks is changed from 7 to 5.

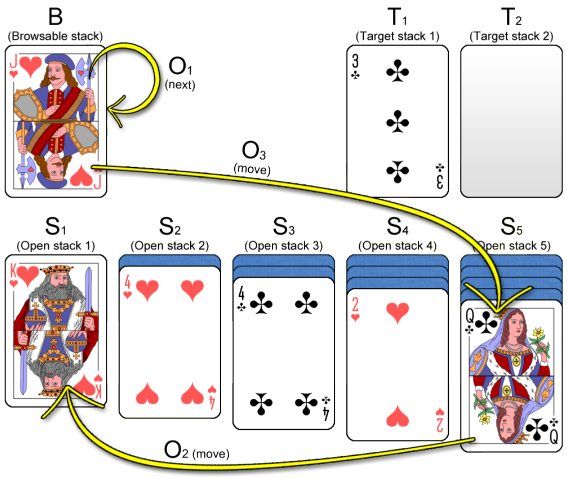

Figure 1: Simplified Solitaire Overview

2.1 Definition

The complete game can formally be defined by the following three sets:

- $A = \{A_♣, ..., K_♣, A_♡, ..., K_♡\}$ is the set of all cards.
- $S = \{B, T_1, T_2, S_1, S_2, S_3, S_4, S_5\}$ is the set of all cardstacks. The cardstacks again are sorted sets of cards. Assume that all cardstacks are sorted from top to bottom. Thus, the first element in each cardstack is the card that can be found on top of this stack.
- $V \subseteq A$ the set of currently visible cards.

In addition, let $X[i]$ refer to the card which can be found at position i in cardstack X starting with $i = 1$ for the card on top of the stack and going on with $i > 1$ towards the bottom of the stack. Furthermore, a configuration $C = (S, V)$ reflects a state of the game. A configuration can be changed by executing an operation. For example a configuration C_1 can be reached from configuration C_0 by execution operation O: $C_0 \xrightarrow{O} C_1$.

In Simplified Solitaire the B stack can be browsed endlessly. The two target cardstacks T_1 and T_2 are used to collect the already sorted cards. In this case the cards have to be sorted consecutively in ascending order from Ace to King and all cards in T_1 and T_2 must have the same suit. All other cardstacks are open stacks which can be used to shift cards around. The cards on these stacks on the other hand have to be ordered consecutively in descending order from King to Ace and the color of each card has to be different from the color of the card below.

Figure 1 shows one possible configuration of a Simplified Solitaire game. In this configuration for example the cardstack $S_2 \in S$ holds the cards $S_2[1] = 4_♡$ and $S_2[2] = 8_♣$. But since the 8 of Clubs is not an element of V, the players cannot see the value of that card. Likewise in Figure 1, T_1 holds the cards $\{3_♣, 2_♣, A_♣\}$ even if only the 3 of Clubs can be seen. Hence, it is part of the goal of the game to reveal all hidden cards in all open stacks $(S_1, ...S_5)$. T_2 is still empty in this example.

[4]Description of the original Solitaire: `https://en.wikipedia.org/wiki/Microsoft_Solitaire`

2.2 Operations

To play the game, every player can access two types of operations, namely `next` and `move`. By executing one of these, the current configuration C_x is transformed into another configuration C_{x+1}. The `next()` operation uncovers the next card on the browsable cardstack (B). Let $x = B[1]$ be the card on top of B and $y = B[2]$ be the card that should be uncovered. At first the `next` operation takes x and puts it at the bottom of B. In turn, y becomes the card on top of B. Secondly, it replaces x by y in V.

The `move(`c`,` $From$`,` To`)` operation moves anything from a single card up to a complete stack on top of another stack. More precisely, it takes c and all cards on top of c in the $From$-stack and puts them on top of the To-stack. Furthermore c is equal to $From[i]$ and if $From[i+1]$ exists and $From[i+1] \notin V$ holds, then $From[i+1]$ is added to V as well as c is added to V if $c \notin V$. In other words, the card below c on the $From$-stack becomes visible.

Considering the example illustrated in Figure 1, the yellow arrows represent three operations O_1, O_2 and O_3. Operation $O_1 =$ `next()` uncovers the next card behind the Jack of Hearts. Denote `move(`$Q_\clubsuit$`,` S_5`,` S_1`)` as operation O_2. Once it is executed the Queen of Clubs and every card on top of it is placed on top of the King of Hearts on stack S_1. By operation $O_3 =$ `move(`$J_\heartsuit$`,` B`,` S_5`)` the Jack of Hearts is placed on top of the Queen of Clubs on stack S_5.

Figure 2 shows the execution of these operations in a two player scenario. Alice and Bob play Simplified Solitaire by collaborating without any extra means of communication. Alice executes operation O_1 (see ①) and immediately reveals the subsequent card in stack B behind the Jack of Hearts. Then Alice moves the Queen of Clubs by executing operation O_2 (see ②). Lastly she receives the operation O_3 from Bob (see ⑤). The occurring problem is that the Queen of Clubs cannot be located on stack S_5 anymore, because the execution of O_2 moved it to S_1. In order to solve this problem, operational transformation can be used. A transformation of operation O_3 makes sure that the Jack of Hearts is still placed on top of the Queen of Clubs. Such an operational transformation is illustrated by the symbol: $\rightarrow_t$. In the current case O_3 is transformed into O_3': $O_3 =$ `move(`$J_\heartsuit$`,` B`,` S_5`)` $\rightarrow_t O_3' =$ `move(`$J_\heartsuit$`,` B`,` S_1`)`. The last parameter has been transformed. By executing O_3' now (see ⑤), the Jack of Hearts is placed on top of the Queen of Clubs on stack S_1 as intended. From Bob's perspective another operation has to be transformed. Once he receives operation O_1 (see ④) he wants to uncover the card under the Jack of Hearts in the browsable stack (B). But this is not required anymore, because the Jack of Hearts has already been moved by his own operation (O_3, ③) what implicitly uncovered the subsequent card. Hence, the operation O_1 is transformed into $O_1' =$ `NOP` (see ④) where `NOP` stands for "no operation" ($O_1 \rightarrow_t O_1'$).

The described transformations are enforced by the collaborative system in order to achieve consistency. How to achieved consistency exactly is defined in the next section based on consistency models.

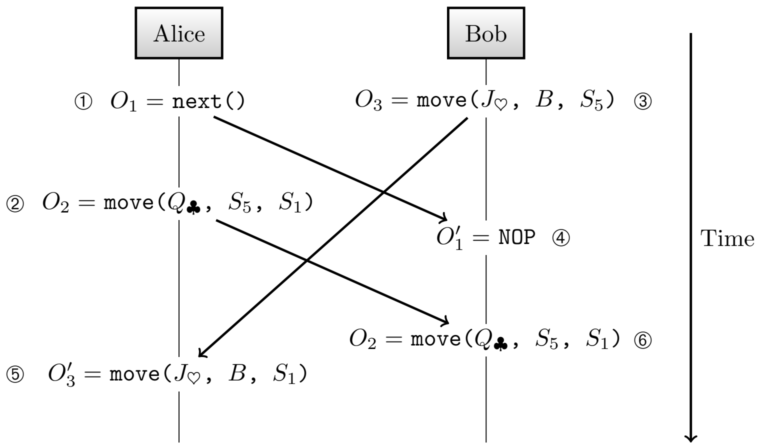
Alice
Bob
① $O_1 =$ next()
$O_3 =$ move($J_\heartsuit$, B, S_5) ③
② $O_2 =$ move($Q_\clubsuit$, S_5, S_1)
$O_1' =$ NOP ④
Time
$O_2 =$ move($Q_\clubsuit$, S_5, S_1) ⑥
⑤ $O_3' =$ move($J_\heartsuit$, B, S_1)

Figure 2: Two Player Execution Scenario

3 Consistency Models

Initially, operational transformation was developed to be used as a part of groupware or collaborative systems. In such a system one of the most important aspects is consistency. Every user should always see and use the same basis, which in the running example would be the board. However, maintaining consistency can be challenging.

To guarantee consistency, the system must follow a *consistency model* that describes exactly how to achieve consistency for all users. Several models have been developed and published to accomplish that task.

In the following, we focus on basic models, because they illustrate best what benefits can be achieved by using operational transformation. Nevertheless, there are other models that might be better in measure of generality and provableness. Furthermore, we discuss how well these models can be applied to games and in particular to Simplified Solitaire.

In general, a consistency model consists of a collection of different properties. A system or game implementing such a model has to make sure that all properties are always satisfied in order to guarantee consistency.

3.1 The CC Model

The CC model is the most basic consistency model and it is also the basis for all other models described in this section. It was introduced in 1989 in the paper "Concurrency Control in Groupware Systems"[1] and is based on two properties, namely *Precedence (**C**ausality)* and ***C**onvergence*. In the following definitions of these properties a *site* refers to a viewpoint of a user. A site in games for example refers to a player's instance of the game.

Definition 1 *The Precedence property ensures that all dependent operations are executed in the same order at all sites.*

One operation is dependent on another if they are related by the precedes relation ($\rightarrow_p$), which is formally defined as Lamport's "happened-before" relation[2]. It relates operations to each other by the time of their generation. Hence, O_a generated at site i precedes O_b generated at site j ($O_a \rightarrow_p O_b$) if and only if one of the following cases is true:

1. $i = j$ and operation O_a was generated before O_b was generated.

2. $i \neq j$ and operation O_a was executed at site j before O_b was generated.

3. There exists a set of operations $\{O_{x_1}, ..., O_{x_n}\}$ with $O_a \rightarrow_p O_{x_1} \rightarrow_p ... \rightarrow_p O_{x_n} \rightarrow_p O_b$.

In the example visualized in Figure 2 only the operation O_2 depends on O_1 ($O_1 \rightarrow_p O_2$), because case 1 is true. Both operations are generated on the same site as well as O_a is generated before O_b. Thereby, O_1 has to be executed before O_2 at all sites. Case 2 is never fulfilled by any set of operations from the example, because Alice and Bob never execute an operation received from the other before generating their own operations. Additionally, since there is only one dependency ($O_1 \rightarrow_p O_2$) in the example, case 3 never holds.

Definition 2 *The Convergence property ensures that the configuration reached after the execution of a set of operations is the same at all sites.*

To achieve the *convergence* property, a total order could be built. This total order could determine for example that any operation O_i with $i < j$ is executed before any operation O_j is executed. For the example described in Section 2, this allows only one execution order. Operation O_1 has to be the first one followed by O_2 which in turn is followed by operation O_3. To achieve such a total order at all sites, an undo/redo scheme could be used. By that, Bob for example would undo operation O_3 once he receives O_1 and then automatically execute O_1 followed by O_3. This method requires that all operations are reversible.

This shows that all properties of the CC Model can be satisfied without using operational transformation at all, but in consequence of that the rules of the game might be violated. For example, by executing O_2 before O_3 the Jack of Hearts is placed on stack S_5 but not on top of the Queen of Clubs as originally intended. Hence, the rules of Simplified Solitaire are violated, because the Jack of Hearts can only be placed on top of the Queen of Clubs or on top of the 10 of Hearts on one of the target stacks. In fact this can happen even if each player by himself only invoked operations that do not violate the rules of Simplified Solitaire. The model which is described next overcomes this issue by adding a further property.

3.2 The CCI Model

This model is built upon the CC model. It consists of three properties: ***C**ausality Preservation (Precedence)*, ***C**onvergence* which are defined in the same way as in the CC model and ***I**ntention Preservation*. To satisfy the latter one a technique like operational transformation has to be used.

Definition 3 *Intention Preservation guarantees that the intention of any executed operation is always the same and independent of the site of execution.*

Assume the initial configuration is C_{Start} and the set of operations that are executed is $\{O_1, O_2, O_3\}$ as depicted in Figure 2. On Alice's site this has to lead to the following configuration changes:

$$C_{Start} \xrightarrow{O_1} C_1 \xrightarrow{O_2} C_2 \xrightarrow{O_3 \rightarrow_t O'_3} C_{Final}$$

On Bob's site on the other hand the sequence of configuration changes has to be:

$$C_{Start} \xrightarrow{O_3} C_3 \xrightarrow{O_1 \rightarrow_t O'_1} C_3 \xrightarrow{O_2} C_{Final}.$$

Unless the operations O_1 and O_3 are transformed as described in Section 2, these sequences violate the rules of Simplified Solitaire. Now to satisfy the intention preservation property, the intention of O'_1 and O'_3 has to match the intention of O_1 and O_3 respectively. O_1 should uncover the next card on stack B. Since that already happened by moving the Jack of Hearts the intention is preserved if O'_1 simply does nothing. The transformation function realizes this by transforming the operation into `NOP`. O_3 should move the Jack of Hearts on top of the Queen of Clubs. After the transformation O'_3 still has the same effect even if the Queen is located on top of another stack, namely S_1. In both cases the transformation preserves the intention of the original operation. So if the intention of any operation is defined according to the rules of Simplified Solitaire this implies that any executed, rule-compliant operation and its transformed replica cannot violate the rules of Solitaire. Additionally, the convergence property is satisfied as well since both players end up with the same configuration (C_{Final}) on their site after execution of all three operations.

Still, the CCI model lacks a formal and generic definition of intentions and therefore does not allow a complete proof of the intention preservation property. However, there exist other models that overcome this problem as mentioned in the next Section.

3.3 Further Consistency Models

There exist at least two more consistency models comparable to the CCI model. In particular, the *CSM model* [3, 4] and the *CA model* [5]. On one hand, these models share the causality property. On the other hand, their other properties are defined differently. But these properties still aim for the same goals as the convergence and intention preservation properties from the CCI model. The only difference is that their definitions are more generic and less ambiguous and by that it is easier to prove them formally. However, since the rules of games already strictly define how to preserve the intention of any operation, the CCI model in combination with these rules is sufficient for maintaining consistency in games like Simplified Solitaire. Hence, the CSM and CA models are not described any further in this paper.

4 Operational Transformation

In order to realize the aforementioned consistency models within a collaborative software system such as Simplified Solitaire, the technique *Operational Transformation* (OT) which was introduced in Section 1 could be used.

When specifying an OT system, an appropriate algorithm must be chosen, called *OT algorithm.* Ellis and Gibbs [1] point out several properties an OT algorithm must have. First, the effect of an operation must be available immediately on the system where it was executed. Second, it must relinquish locking techniques and third, it must be fully distributive. The latter means, that the OT algorithm must be able to handle *concurrent* operations generated on multiple *sites* where a site refers to a user's local system on which an instance of the OT algorithm runs. Two operations are considered concurrent if they were simultaneously generated on different sites based on the same configuration. Concurrent operations are *potentially conflicted* as they potentially lead to different configurations on each site after being executed. This is the case if another operation was executed in between. Therefore, the OT algorithm chooses an appropriate *OT function* according to the operations' type. Such an OT function checks whether the operations are in conflict and transforms one of them accordingly. However, if no conflict is detected, no transformation is performed and the operation is executed directly.

In the following, we introduce the properties an appropriate OT algorithm must fulfil in order to meet the requirements of the CCI model which was considered suitable for collaborative games in Subsection 3.2. We also present exemplary OT function definitions and show their use based on our running example introduced in Section 2.

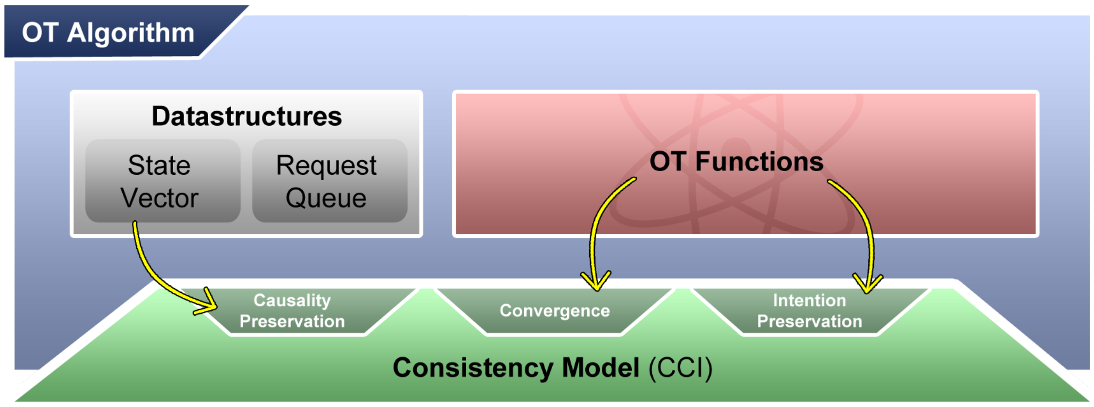

Figure 3: Overview OT Algorithm and Consistency Model

4.1 OT Algorithm

The OT algorithm and the different involved components described in this section are depicted in Figure 3. Such an algorithm must be able to guarantee the following three properties defined for the CCI model: Causality Preservation (by using *state vectors*), Convergence and Intention Preservation (by using *OT functions*). As will be explained in more detail, these properties can be guaranteed by implementing specific data structures and programmable constructs called *OT functions*. We show how it is possible to achieve these goals using techniques originally introduced by the *dOPT*[5] algorithm [1].

Causality preservation implies that the execution order of incoming dependent operations must be preserved. dOPT solves this issue by maintaining a *state vector*. The state vector is a data structure represented as an n-tuple $(s_1, ..., s_n)$ where s_i is the number of operations executed by site i. In the running example there are $n = 2$ players where s_1 would be the number of executed operations generated on Alice's site and s_2 those of Bob. When an operation O is executed at site i, it is broadcast carrying i's current state vector. Afterwards it updates the state vector of the sending site. Sending site i's vector along with O helps the receiving sites to determine whether an operation is received in the correct order as will be illustrated in the following paragraph. We denote the state vector delivered by Operation O as $SV_O(s_1, ..., s_n)$, whereas $SV_i(t_1, ..., t_n)$ denotes the current state vector of site i. When O is received at a site, the algorithm determines if it can be executed directly, if it must be transformed first or if it cannot be executed at the particular time and therefore must wait for its processing. An example for the latter case is given in the following. If O or its transformation is executed, it updates the site's state vector accordingly by incrementing the generating site's element by one. We denote that as $O \rightarrow_{SV} (s_1, ..., s_n)$.

In context of the running example, Figure 4 depicts two situations and how using a state vector helps preserve the causality of operations. We assume that $O_1 \rightarrow_p O_2$ and that all state vectors are initialised with $(0, 0)$ meaning that no operation of Alice ($i = 1$) and also none of Bob ($i = 2$) was executed yet. In Figure 4a, the algorithm on Bob's site first executes O_3 (see ⑦) and then broadcasts the operation along with its current state vector of $(0, 0)$ before updating it to $(0, 1)$. On arrival of O_1 and O_2 (see ⑧ and ⑨) it would find that they were received in the correct order and subsequently update Bob's current state vector to $(1, 1)$ and $(2, 1)$. Eventually, Bob's current state vector of $(2, 1)$ expresses that Bob's algorithm has executed two operations of Alice and one operation of Bob.

In Figure 4b Alice executes O_1 and sends it to Bob along with her current state vector of $(0, 0)$ (see ❶) before updating her state vector to $(1, 0)$. Next, she executes and sends operation O_2 (see ❷). Even though both O_1 and O_2 are sent subsequently, assume in this example, that due to network issues they arrive in a different order at Bob's site (see ❸ and ❹). On reception of O_2 (see ❸), Bob's algorithms, would find that O_2 must have been received out of order since the state vector $(1, 0)$ expresses that Alice has already executed one operation generated at her site (see ❶) while Bob's state vector is still $(0, 1)$, since Bob has not yet executed an operation of Alice. Consequently, Bob's algorithm would wait for the arrival of O_1 (see ❹) before processing O_2. Without sending the state vector along with the operations, Bob would not be able to detect operations that arrive out of order and consequently violate causality when executing them directly in the order they were received.

[5]dOPT: Distributed Operational Transformation

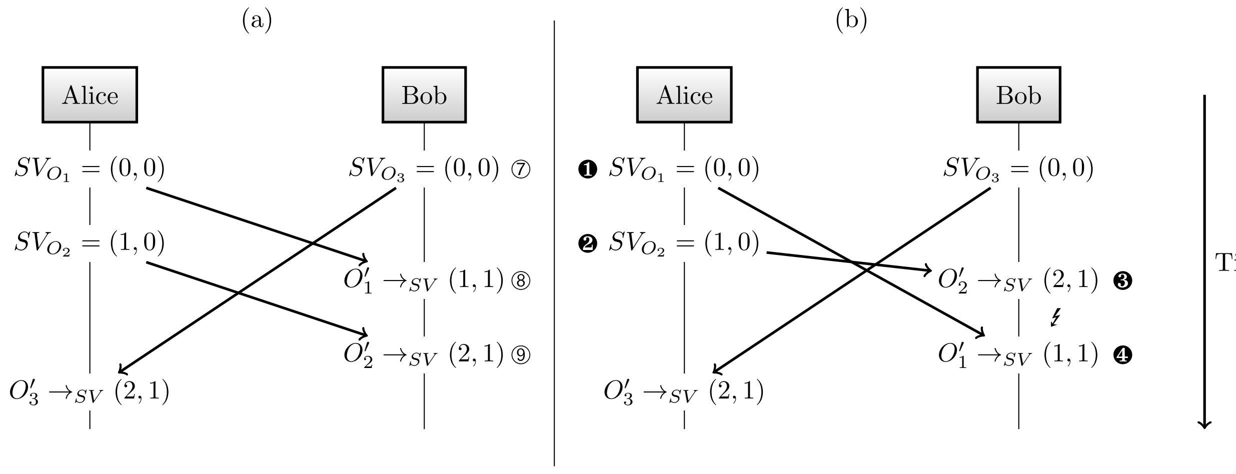
(a)
Alice
Bob
$SV_{O_1} = (0,0)$
$SV_{O_3} = (0,0)$ ⑦
$SV_{O_2} = (1,0)$
$O'_1 \rightarrow_{SV} (1,1)$ ⑧
$O'_2 \rightarrow_{SV} (2,1)$ ⑨
$O'_3 \rightarrow_{SV} (2,1)$
(b)
Alice
Bob
❶ $SV_{O_1} = (0,0)$
$SV_{O_3} = (0,0)$
❷ $SV_{O_2} = (1,0)$
$O'_2 \rightarrow_{SV} (2,1)$ ❸
$O'_1 \rightarrow_{SV} (1,1)$ ❹
$O'_3 \rightarrow_{SV} (2,1)$
Time

Figure 4: Using a state vector to preserve causality.

To guarantee convergence and intention preservation it does not suffice to only reorder the execution of incoming operations [6]. Under the circumstances described in the following, it might be necessary to transform a received operation against an already executed operation using an appropriate *OT function*. OT functions are described further in Subsection 4.2. Intentions are preserved by transforming the operation according to the game's rules. Consequently, the intention of a transformed operation is equal to its original one. For instance, regarding the running example, Bob intends to place the Jack of Hearts on the Queen of Clubs (O_3, ③). Since Alice moved the Queen of Clubs to another stack (see ②), Bob's untransformed operation would become invalid at Alice's site, according to the rules. Thus, O_3 must be transformed such that the Jack of Hearts is, as intended, placed on the Queen of Hearts (O'_3, ⑤), which both revalidates O_3 and preserves its intention.

The task of choosing which operation shall be executed next and determining against which already executed operations it must be transformed is crucial. dOPT solves this task with the help of a data structure called *request queue*[6], which contains all operations awaiting execution at a site i. The algorithm running on site i searches this queue and for each operation O_j the corresponding state vector s_j is compared to site i's current state vector s_i. Hence, three cases must be considered:

1. $SV_{O_j}(s_1, ..., s_n) = SV_i(t_1, ..., t_n)$: Operation O_j can be executed directly without any transformation since both sites i and j have executed the same number of operations for each site and as a result do not work on inconsistent configurations at that point.

2. $SV_{O_j}(s_1, ..., s_n) \neq SV_i(t_1, ..., t_n) \wedge \exists s_k, t_k : s_k > t_k,\ k \in \{1, ..., n\}$: Operation O_j cannot be executed as site j has executed at least one operation more than site i.

3. Else: O_j can be executed but is conflicted with at least one formerly executed operation. It must consequently be transformed first since SV_{O_j} refers to a state older than SV_i. Next it must be determined which of the operations have already been executed on site i but not yet on site j when O_j was generated. These operations have not affected O_j and are therefore conflicted with it. Consequently, they must be taken into account for O_j's transformation.

By applying the above comparisons and choosing an appropriate OT function for case 3, the OT algorithm is able to ensure that after execution of all, possibly transformed, operations on each site, all sites share equal configurations, thus guaranteeing convergence.

4.2 OT Functions

If an operation must be transformed against another one, the OT algorithm chooses an appropriate OT function according to their respective operation types. Simplified Solitaire supports two operation types: `move` and `next`. Hence, there must exist OT functions for every combination of the two types. The algorithm dOPT realizes that by building an $n \times n$ *transformation matrix* where n is the number of available operations and each element is an OT function denoted as $T_{<type_{op_1}>,<type_{op_2}>}$.

[6]The request queue needs not necessarily be implemented as a classical First-in-first-out queue.

The 2×2 transformation matrix for Simplified Solitaire would look as follows:

	move	next
move	$T_{move,move}$	$T_{move,next}$
next	$T_{next,move}$	$T_{next,next}$

In the running example depicted in Figure 2, on arrival of operation O_1 (see ④), Bob's algorithm would detect that O_1 carries the vector $(0, 0)$ while Bob's current vector is $(0, 1)$ and therefore case 3 as described in the previous section would apply. This situation is also illustrated in Figure 4a (see ⑧). It would further find that O_3 is the only operation that was executed on Bob's site (see ⑦) but not on Alice's and therefore transform O_1 against O_3. Next, it must choose the appropriate OT function. Since O_1 is of type `next` and O_3 is of type `move`, it would choose $T_{next,move}$ as an OT function where the first parameter is O_1 and the second O_3. An example call could be:

O_1' `:=` T$_{next,move}$`(next(), move(`$J_\heartsuit$`, `B`, `S_5`))`

Since a `next` operation implicitly always affects the browsable stack B, it does not need any parameters. The first parameter of a `move` operation is the card that is to be moved ($J_\heartsuit$). The second parameter is the stack, the card is removed from (B) and the third parameter is the stack, the card is moved to (S_5).

The function could be implemented as follows:

```
1 T$_{next,move}$(next(), move(card, from, to)):
2   if (from = $B$)
3     return NOP // see $O_1'$
4
5   return next() // default: no conflict
```

Listing 1: Example implementation of a transformation function for a next/move operation pair.

The function $T_{next,move}$ takes two operations as arguments where the first operation argument is transformed against the second. Knowing that the first operation unveils the next card on the browsable stack B and in turn covers the former topmost card, the function must check whether the second operation intends to move that card (Listing 1, Line 2). If this is the case, the first operation is transformed into a `NOP` operation (Listing 1, Line 3) which has no effect on Bob's configuration. The reason for the correctness of this transformation is that moving the topmost card of B has the same effect as executing a `next` operation and by that, the transformation preserves the intention of the original operation. Note, that strictly speaking, the configurations of Alice and Bob are not equal after execution of the transformed operation O_1' (see ④) as Alice would expect the Jack of Hearts still lie covered on B in her configuration. Hence, the transformation on Alice's site must take this into account in order to guarantee convergence.

When O_3 arrives at Alice's site (see ⑤), it is added to Alice's request queue - in which it is the only element. O_1 and O_2 have already been subsequently executed on Alice's site (see ① and ②), thus Alice's current state vector would be $(2, 0)$. The algorithm now compares this vector to that of O_3 which is $(0, 1)$. Again, case 3 applies and the algorithm searches for appropriate transformation candidates. Since, according to O_3's state vector, both O_1 and O_2 have not yet been executed at Bob's site when O_3 was generated (see ③), O_3 must first be transformed against the more recent O_2 and then against O_1. The following statements could exemplarily be executed:

O_3' `:=` T$_{move,move}$`(move(`$J_\heartsuit$`, `B`, `S_5`), move(`$Q_\clubsuit$`, `S_5`, `S_1`))`
O_3'' `:=` T$_{move,next}$`(`O_3'`=move(`$J_\heartsuit$`, `B`, `S_1`), next())`

The relevant OT functions could be implemented as in Listings 2 and 3.

Since the number of cases that need to be checked can easily grow very large, we focus only on a subset of all conflicting cases that can be found in our examples. In addition, the above implementations of OT functions are by no means optimized as they serve primarily as comprehensible examples.

In Listing 2, the function $T_{move,move}$ checks in Line 2 whether the destination stack of the first operation (`to1`) is equal to the source stack of the second (`from2`). If this is the case, the operation which is to be transformed seemingly intends to move a card onto another, which has moved to another position. If this position is not a target stack, the function replaces the first operation's destination (`to1`) with that of the second operation (`to2`) and returns the transformed operation (Line 3). As a result, the intention of Bob, which is moving the Jack of Hearts onto the Queen of Clubs, can be preserved. As explained before, following the first transformation, the

```
1  T_{move,move}(move(card1, from1, to1), move(card2, from2, to2)):
2    if (to1 = from2 and to2 ∉ {T1,T2})
3      return move(card1, from1, to2) // see O'2
4
5    if (to1 = from2 and from2 ∈ {T1,T2})
6      return [move(card1, to1,   from1), // see O'5
7              move(card2, from2, to2)]
8
9    if (to2 = from1 and from1 ∈ {T1,T2})
10     return NOP // see O'4
11
12   [...]
13
14   return move(card1, from1, to1) // default: no conflict
```

Listing 2: Example implementations of move/move and move/next transformation functions.

```
1  T_{move,next}(move(card, from, to), next()):
2    if (from = B)
3       return move(card, from, to) // see O'3
4
5    [...]
6
7    move(card, from, to) // default: no conflict
```

Listing 3: Example implementations of move/move and move/next transformation functions.

result (O'_3) needs to be transformed against O_1 (Listing 3, Lines 2-3). If the `move` operation intends to move a card from the browsable stack B while the same card is about to be covered by the `next` operation, the effect on the configuration is the same for both. Thus, no further transformation must be performed.

When O_2 arrives at Bob's site (see ⑨), it carries the state vector $(1, 0)$. Since O'_1 was executed before, Bob's current vector is $(1, 1)$. Hence, O_2 must be transformed against O'_1 and O_3. Both OT functions will find that there is no conflict with O'_1 and O_3 respectively and thus execute O_2 as is (see ⑥). Executing an operation without transformation is the default for all OT functions since it often suffices in order to achieve convergence.

Up to this point, we did not explicitly mention any prioritizations. For games, the most logical highest priority is: reaching the goal. Thus, this priority must be considered when defining an OT function even if that implies that an operation needs in turn be undone.

In the following, we discuss a special case where an operation cannot be transformed directly and must be undone first. Figure 5 depicts a scenario where Alice removes a card from the same target stack Bob tries to place a card on.

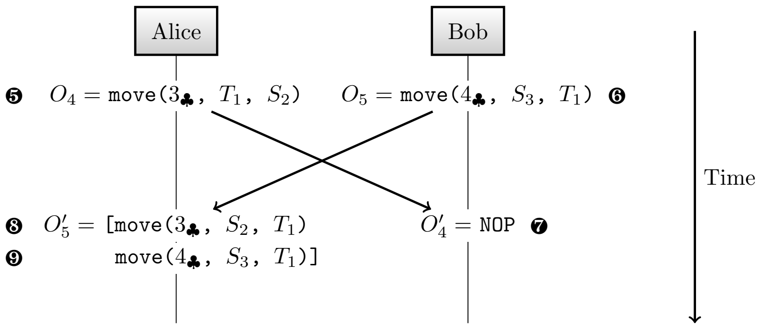
Alice
Bob
❺ O_4 = move(3♣, T_1, S_2)
O_5 = move(4♣, S_3, T_1) ❻
Time
❽ O'_5 = [move(3♣, S_2, T_1)
❾ move(4♣, S_3, T_1)]
O'_4 = NOP ❼

Figure 5: Special conflict case.

Alice intends to move the Three of Clubs from target stack T_1 onto open stack S_2 (O_4, ❺). Simultaneously, Bob moves the Four of Clubs to T_1 (O_5, ❻). This obviously is a conflict because Bob's move would not be valid according to the rules if Alice's move is executed and vice versa. Thus, a prioritization must be defined. As stated above, the priority in games is usually reaching the game's goal. The goal of Simplified Solitaire is reached if all cards are placed on both target stacks, one for each color. Thus, moving a card to one of the target stacks is assigned a higher priority than removing a card from a target stack. This also solves conflicts where two players move the same card to different stacks, since a card can either be placed on exactly one target stack or one open stack.

See Listing 2 for an exemplary implementation of the transformations of O_4 and O_5 (Lines 10 and 6-7). The respective function calls could look as follows:

$$O'_4 \ \texttt{:=}\ \texttt{T}_{move,move}\texttt{(move(}3_{\clubsuit}\texttt{, } T_1\texttt{, } S_2\texttt{), move(}4_{\clubsuit}\texttt{, } S_3\texttt{, } T_1\texttt{))}$$
$$O'_5 \ \texttt{:=}\ \texttt{T}_{move,move}\texttt{(move(}4_{\clubsuit}\texttt{, } S_3\texttt{, } T_1\texttt{), move(}3_{\clubsuit}\texttt{, } T_1\texttt{, } S_2\texttt{))}$$

When O_4 arrives at Bob's site (see ❼), it needs to be transformed against O_5. Since both operations are contrary in that one operation turns the other invalid and O_5 has a higher priority, O_4 is transformed into a `NOP` (Listing 1, Line 10) which makes O_4 have no effect on Bob's configuration. On the other side, Alice receives O_5 (see ❽) after already having executed O_4 (see ❺) and the OT algorithm finds that O_5 needs to be transformed against O_4. Again, as O_5 has higher priority than O_4 and both operations are contrary, O_4 must first be undone (Listing 1, Line 6). The configuration after execution of this operation is the same as the one at the time O_5 was generated at Bob's site (see ❻). Thus, O_5 can be executed directly (Listing 1, Line 7, ❾), leading to equal configurations on both Alice's and Bob's site. Again, note that by applying an OT function the system reached a convergent state with Bob's intention preserved. Even though Alice's original intention was violated because her operation was simply undone, we reason that due to sensible prioritization towards the goal of the game, Alice's move is made invalid and would therefore not be executed anyway.

Sun et. al. distinguish between two types of transformation functions: *inclusion* and *exclusion* transformation [7]. Given two operations O_i and O_j, an inclusion transformation transforms O_i against O_j explicitly including the impact of O_j. An exclusion operation on the other hand excludes the impact of O_j thus preserving the intention for operation O_i by omitting the impact of O_j. Even though it is not always possible to clearly distinguish between inclusion and exclusion transformation, the above defined OT functions can be considered inclusion transformations.

Exclusion transformations were originally developed to overcome the major drawback of dOPT which is the lack of guaranteeing intention preservation. This makes dOPT primarily applicable for the CC model described in Subsection 3.1. To overcome this drawback, the *adOPTed* algorithm was developed [8]. adOPTed applies an exclusion transformation first if necessary to undo the impact of a formerly executed operation. Afterwards, an inclusion transformation can be applied to reach a convergent state. However, such exclusion transformations are not necessary for Simplified Solitaire since our algorithm is able to achieve intention preservation and are therefore not further explored here. Apart from dOPT and adOPTed, there currently exists a plethora of different OT algorithms [9] - some of which make use of exclusion transformations and some do not.

In the section above, we showed which requirements an algorithm has to meet in order to realize the properties of the CCI model. Apart from guaranteeing causality preservation by processing incoming operations in a sensible order, it must be capable of determining which operations must be transformed against each other in order to guarantee intention preservation which in turn leads to convergence. We showed exemplary implementations of transformation functions and how the priority of reaching the goal of the game can resolve conflicts between operations where one invalidates the other when executed.

The latter is an example of how simple it is to find adequate priorities what makes operational transformation particularly suitable for games. Apart from this benefit, in the following, we describe arguments supporting the use of operational transformation in games.

5 Relevance in Games

Above we have shown that a game like Simplified Solitaire can be played in a collaborative way. Thus, the original Solitaire can be collaboratively playable as well. However, there would be more special cases that have to be taken into account in the OT functions. Solitaire can be categorized as a card-puzzle game but it also fits into other categories. For example Solitaire is a *multi-object* game since it considers multiple cards and stacks at the same time. *Single-object* games accordingly would be all the games that only focus on one object, e.g. racing games where the player only controls a single vehicle. Additionally, Solitaire is not a *turn-based* game since the players do not have to wait for an opponent after making a turn. A game such as chess on the other hand is turn-based. After each turn every player has to wait for the turn of the other player. In Figure 6 these categories are used to compare games and their ability to be played collaboratively. Each set of games is rated accordingly to its ability to be played in a collaborative way. Furthermore, the figure contains some examples for games of each category.

Multi-object, not turn-based games such as Solitaire can be found in the upper right corner. These games seem to be most promising as stated by the + in the figure. The reason is that multiple players can invoke different operations which may be merged into a single action through techniques like operational transformation. This may give two or more collaborative players an advantage over a single player. Considering the running example Alice moves the Queen of Clubs and Bob the Jack of Hearts. With operational transformation both moves get merged into a single action. A single player in contrast has to do one move after another. The observable advantage in such a situation is that the two collaborative players can be faster.

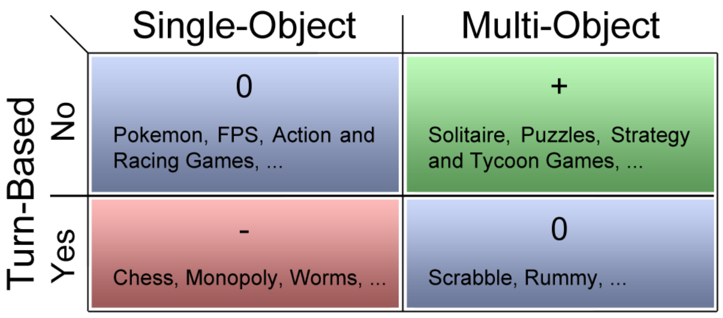
Single-Object
Multi-Object
Turn-Based
No
Yes
0
Pokemon, FPS, Action and Racing Games, ...
+
Solitaire, Puzzles, Strategy and Tycoon Games, ...
-
Chess, Monopoly, Worms, ...
0
Scrabble, Rummy, ...

Figure 6: Eligibility of games for collaboration

In single-object games this advantage can easily turn into a disadvantage. Considering the Pokémon example introduced in the introduction (see Section 1), the Pokémon trainer (single controlled object) could for instance make a horizontal or vertical step. If Alice intends to move rightwards whereas Bob wants to move upwards, these two turns can be merged into a diagonal step[7]. On the other hand, if Alice still intends to move rightwards whereas Bob wants to move leftwards, these two turns cannot be merged reasonably. Anyhow, this can happen in multi-object games as well. For example, in Simplified Solitaire two players could simply want to move the same card to different places. But since the players most likely have more choices, we assume that it does not occur as often as in single-object games. A more detailed description of a similar problem and its solution in Simplified Solitaire is described in Section 4.2. Priorities are used to decide which action is executed in the end. Hence, the advantages of playing a single-object game collaboratively are limited. We state that by the 0 in Figure 6.

Turn-based, multi-object games can be played collaboratively as well but only if each turn is played collaboratively by a team instead of a single player. For example Scrabble[8] is played by two teams against each other and each team consists of at least two players. In this situation each team can collaborate while constructing a word with the letters of their team. This process can be enhanced with techniques like operational transformation to gain the same advantages as described above. But once a single player ends their team's turn, no more changes can be applied by anyone. Considering operational transformation this may lead to further problems. An operation that has been generated but was not executed at all sites before the turn ends represents one situation that could lead to such problems. Because of that, we consider this category of games as partly collaboratively playable again stated by the 0 in the figure.

Lastly, all games that are turn-based and only consider a single object can never profit from being played in a collaborative way. Since only one action can be done per turn, it is impossible to apply a technique such as operational transformation. Considering chess for example, once a player has made a move the turn ends immediately. Hence, in no situation two operations can be merged in order to generate an advantage.

Games such as Simplified Solitaire could open up a new category of games, namely *collaborative games*. These should not be mixed up with *cooperative games*. In cooperative games multiple players try to achieve one goal. To do so, most of these games split up the main goal into subgoals. Considering Simplified Solitaire, Alice for instance could focus on putting all cards onto the target stacks whereas Bob tries to uncover all hidden cards. Since each player over time may become a specialist in her or his task, this might lead to an advantage over a single player. In collaborative games the main goal is typically not subdivided. Every player attempts to reach the main goal and in doing so the collaborative system itself leads to the advantage over non-collaborating players for example by transforming or merging operations. This definition of collaborative games and its distinction from cooperative games is only justified by the findings in this paper. Furthermore, while both terms are often used as synonyms there also exist other definitions. For instance, in a game theory paper different aspects are considered [10]: Cooperative games are distinct from collaborative games by the fact that all players of a cooperative game get individual scores for reaching their subgoals whereas players of a collaborative game all

[7]Simplified example: Diagonal steps are not possible in the original Pokémon game.
[8]`http://scrabble.hasbro.com`

get the same score for reaching the main goal. We do not consider scoring at all in this paper but still both definitions have in common that the main goal is split into subgoals when it comes to cooperative games. Hence, both definitions could be merged.

As a last remark, we state that collaborative games can be developed by using existing operational transformation frameworks. This reduces the effort to create collaborative games significantly. An implementation of a collaborative game was presented by Google[9]: Multiple players could try to solve one Rubik's cube. For the implementation they used the Google Realtime API[10] which is based on operational transformation. Nevertheless, most games do not consider collaborative play at all. One game that allows collaborative play is Age of Empires II HD[11], which is a strategy game in which a single army can be controlled by multiple players. However, since it does not contain any techniques like operational transformation that feature is not advantageous unless the players themselves come up with subgoals to organize their play (e.g. Alice controls the army whereas Bob handles the reinforcement production). But per definition this makes it a cooperative game.

6 Conclusion

In this paper, we have explained operational transformation in context of games and presented how an OT algorithm could be designed to guarantee properties of a consistency model. Simplified Solitaire has been used as a case-study to show that operational transformation can be used in games as well as in any other collaborative systems. Furthermore, we discussed the advantages and disadvantages of operational transformation in games. In the end, we conclude that operational transformation can be beneficial in games of a specific category which we call collaborative games (e.g. Simplified Solitaire). It can be considered as further work to find out if there are other or better approaches to implement collaborative games.

In the introduction we mentioned Twitch plays Pokémon, an experiment that allowed a million users to play the same game over Twitch[12]. It turned out that the Pokémon games are not the best games to be played collaboratively, although Twitch plays Pokémon shows that people enjoy playing games together in a collaborative way. Hence, the number of collaborative games will probably grow in future and operational transformation is one approach which could facilitate the collaborative aspect of these games.

[9]A presentation by Brian Cairns and Cheryl Simon in 2013 during a Google Developers I/O session: `https://www.youtube.com/watch?v=hv14PTbkIs0`

[10]`https://developers.google.com/google-apps/realtime/overview`

[11]`https://www.ageofempires.com`

[12]Streaming platform: `https://www.twitch.tv/`

References

[1] C. A. Ellis and S. J. Gibbs, “Concurrency control in groupware systems,” in *ACM SIGMOD Record*, vol. 18, pp. 399–407, ACM, 1989.

[2] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[3] D. Li and R. Li, “Preserving operation effects relation in group editors,” in *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work (CSCW)*, pp. 457–466, ACM, 2004.

[4] R. Li and D. Li, “A new operational transformation framework for real-time group editors,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 3, pp. 307–319, 2007.

[5] R. Li and D. Li, “Commutativity-based concurrency control in groupware,” in *2005 International Conference on Collaborative Computing: Networking, Applications and Worksharing*, p. 10ff., 2005.

[6] C. Sun, Y. Zhang, X. Jia, and Y. Yang, “A generic operation transformation scheme for consistency maintenance in real-time cooperative editing systems,” in *Proceedings of the international ACM SIGGROUP conference on Supporting group work: the integration challenge*, pp. 425–434, ACM, 1997.

[7] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, “Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems,” *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 5, no. 1, pp. 63–108, 1998.

[8] D. N.-R. Matthias Ressel and R. Gunzenhuser, “An integrating, transformation-oriented approach to concurrency control and undo in group editors.,” in *Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pp. 288–297, ACM, 1996.

[9] S. Kumawat and A. Khuneta, “A survey on operational transformation algorithms: Challenges, issues and achievements,” *International Journal of Computer Applications*, vol. 3, no. 12, pp. 30–38, 2010.

[10] J. P. Zagal, J. Rick, and I. Hsi, “Collaborative games: Lessons learned from board games,” *Simul. Gaming*, vol. 37, no. 1, pp. 24–40, 2006.