Operational Transformation for Dependency Conflict Resolution in Real-time Collaborative 3D Design Systems

Agustina, Chengzheng Sun, Dong Xu
School of Computer Engineering, Nanyang Technological University, Singapore
{Agustina, CZSun, DongXu}@ntu.edu.sg

ABSTRACT
Conflict resolution is one major challenge in real-time distributed collaborative 3D design systems, which allow concurrent collaborative work on shared 3D documents. Operational Transformation (OT) is a core conflict resolution technique in a range of real-world collaborative systems. No existing OT technique is, however, capable of resolving conflicts among objects with dependency relations, i.e. an update of one object may propagate to other connected/dependent objects, which is commonly used in 3D or complex graphic design systems. This paper contributes a novel OT solution with such a capability for collaborative 3D design systems. This work is the first to extend OT capability to dependency conflict resolution, and OT application scope from 1D/2D to 3D applications, thus advancing the state-of-the-art of OT in both theory and practical application. The proposed solution has been theoretically verified for its correctness in detecting dependency conflicts and achieving consistent results, and implemented in the CoMaya collaborative design system.

Author Keywords
Concurrency control; consistency maintenance; dependency conflict resolution; operational transformation; real-time collaborative 3D design systems.

ACM Classification Keywords
H.5.3 [**Information Interfaces and Presentation**]: Group and Organization Interfaces – Synchronous Interaction.

General Terms
Algorithms; Theory.

INTRODUCTION
The process of creating 3D digital media contents for movies, scientific explorations, product simulations, etc. is very complex and lengthy. It typically involves collaboration from a group of designers with different roles or expertise and possibly at geographically dispersed locations. This collaboration process can be supported by real-time collaborative 3D design systems, which allow multiple geographically dispersed people to edit shared 3D documents at the same time over computer networks [1]. Such systems will increase not only the efficient use of human and financial resources, but also the productiveness, quality, and competitiveness of the work of 3D designers.

To achieve good responsiveness in Internet environments, nearly all real-time collaborative editing systems [1,2,8,9,21] adopt a *replicated* architecture, i.e. shared documents and the application are replicated at all sites in a collaborating session. Advanced collaborative editing systems support the notion of *unconstrained* collaboration [17], i.e. multiple users can *freely* and *concurrently* interact with the local copy of the application and document. One core issue in supporting unconstrained collaboration over distributed and replicated documents is conflict resolution in the face of concurrent operations [4,6,15,17].

In collaborative document editing, two operations conflict if they are concurrent and their executions in different orders result in inconsistent document states. Conflicts can be prevented by prohibiting concurrency (e.g. by locking or floor-control) [3,6,10], or resolved by concurrency control techniques (e.g. serialization [7,8,22] and *Operational Transformation* (OT) [4,16,17,18,20]). Early collaborative 3D (virtual environments) systems adopt either locking (e.g. in blue-C [11], Mu3D [5], etc.), or serialization (e.g. in Avocado [22], DIV [7], etc.) for achieving document consistency. Prohibiting concurrency contradicts with the objective of unconstrained collaboration, thus it is not an option for advanced real-time collaborative editors [21]. Serialization allows concurrency and is able to ensure document *convergence*, but unable to achieve *intention preservation* [17]. OT, on the other hand, is able to resolve conflicts and maintain consistency (achieving both *convergence* and *intention preservation*) in unconstrained collaborative environments. OT was originally invented for supporting collaborative editing of plain text documents [4], and was later extended for supporting any linearly addressable objects (e.g. office productivity documents) and for resolving conflicts among concurrent operations on overlapping objects [21]. OT has been increasingly adopted as the core real-time collaboration technique by industrial collaborative applications, including SubEthaEdit[1], Google Wave/Docs[2], IBM OpenCoWeb[3], and Codoxware[4].

CSCW'12, February 11–15, 2012, Seattle, Washington, USA.

[1] http://www.codingmonkeys.de/subethaedit/
[2] http://en.wikipedia.org/wiki/Google_Docs
[3] https://github.com/opencoweb/coweb#readme
[4] http://www.codoxware.com

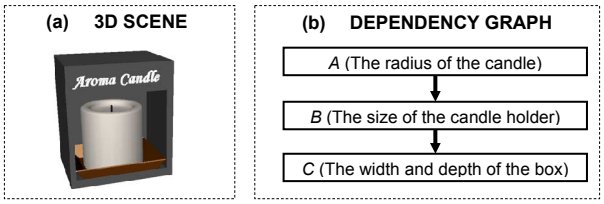

Figure 2. Dependency relations of an aroma candle.

From the point of view of conflict resolution, state-of-the-art OT techniques are capable of resolving conflicts caused by concurrent operations on: (1) *address-dependent* objects (e.g. concurrent insert and delete on a sequence of characters [16]); and (2) *overlapping* objects (e.g. concurrent updates on the same attribute of the same object [21]). However, existing OT techniques are restricted to supporting documents with *attribute-independent* data objects, i.e. an update of the attribute value of one object has no impact on the attribute value of another object. The attribute-independent object model is suitable for plain or rich text documents for office productivity tools (e.g. Word, PowerPoint, etc.), but unsuitable for complex graphic documents (e.g. 3D documents supported by design tools like Autodesk Maya, 3ds Max, etc.)

To support comprehensive 3D graphic functionalities (e.g. modeling, rigging, animation, deformation, etc.), professional 3D design applications map high-level 3D objects into internal graphic objects (called *g-objects*), and organize them into a *Dependency Graph* (*DG*) [13,22]. Many 3D design applications provide multiple user interfaces (UIs) to allow end-users to manipulate 3D-objects (which are then mapped into operations on internal g-objects), and directly manipulate the internal g-objects. In a DG, nodes represent g-objects, and arrows represent *attribute-dependency* relationships among connected g-objects: when the attribute of one g-object is changed (by any means), the changed attribute value will propagate from this g-object to all downstream g-objects (if any) in the DG.

For example, the 3D model of an aroma candle in Figure 1a has, among others, three internal g-objects with the attribute dependency relationships as shown in Figure 1b: g-object *C* (representing the width and depth of the box) depends on g-object *B* (representing the size of the candle holder) that depends on g-object *A* (representing the radius of the candle). Under such dependency relationships, a change to the radius of the candle (by a user from the 3D or g-object UI) will also change the size of the holder and then the width and depth of the box. These dependencies capture the interdependent relationships among related objects and are useful in fitting the candle to the candle holder and box. However, with such attribute-dependency relationships among g-objects, concurrent operations may conflict with each other even if their target g-objects are not overlapping.

To illustrate this special conflict problem in collaborative DG-based 3D design systems, consider the concurrency scenario using the 3D model in Figure 1. Suppose two users intend to modify different aspects of the model

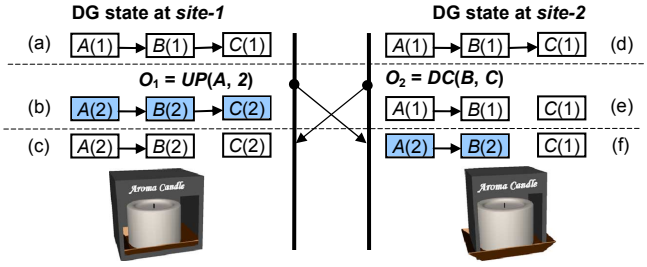

Figure 1. Dependency conflict in a DG.

concurrently: one is to make the candle fatter, and the other is to change the box design for containing multiple candles and to this end, this user has to first remove the dependency relation of the box with the candle holder. Although the two users are modifying different aspects of the same model, their actions happen to be conflicting due to the internal dependency relation of the targets, as explained below. Let $N(x)$ denotes node N with an attribute value x and $\rightarrow$ denotes a connection (attribute-dependency relationship), the initial DG state is: $A(1)\rightarrow B(1)\rightarrow C(1)$ (Figure 2a and d). To achieve their purposes, the two users generate two operations concurrently: O_1 is to increase the radius of the candle by updating the attribute value of node A from $A(1)$ to $A(2)$, which is propagated to downstream nodes, and O_2 is to remove the dependency of the candle holder with the candle by removing the connection between nodes B and C. After executing the two operations in different orders: (1) O_1 followed by O_2 (at *site-1*), the DG becomes $A(2)\rightarrow B(2)$ and $C(2)$ (Figure 2c), with the candle and its holder still fit into the box; and (2) O_2 followed by O_1 (at *site-2*), the DG becomes $A(2)\rightarrow B(2)$ and $C(1)$ (Figure 2f), with the candle holder no longer fits to the box. Clearly, the two candle models are different due to inconsistent final DGs: $C(2) \neq C(1)$. This means O_1 and O_2 conflict with each other even though they do not share any common target object! Their conflict is due to the attribute-dependency relationship among their target objects: nodes B and C (targets of O_2) are depending on node A (the target of O_1). This kind of conflict is named as *Dependency Conflict.*

In unconstrained collaborative editing environments, operations with arbitrary *causal-dependency* (or concurrency) relations can be generated to target objects with arbitrary *attribute-dependency* relationships. Conflict resolution and consistency maintenance (achieving both convergence and intention preservation) in such environments is a new challenge that has never been addressed by prior work [1,5,8,11,15,17]. In this paper, we contribute a novel OT technique for dependency conflict resolution in collaborative DG-based 3D design systems.

The rest of the paper is organized as follows. First, we define the basic object and operation models in a DG. We then discuss the dependency conflict relations and detection conditions, and define the combined effects for conflict and compatible operations. Afterwards, we present the OT solution for dependency conflict resolution. Finally, we summarize the main contributions and future work.

OBJECTS AND OPERATIONS IN A DG

Object Dependency Relations

A DG is composed of a collection of nodes (representing g-objects) and arrows (representing attribute-dependency relations among g-objects), as defined below.

DEFINITION 1. Dependency Graph. *A dependency graph DG is represented as a tuple* $(\mathcal{N}, \mathcal{A})$*, where* $\mathcal{N}$ *is a set of nodes, and* $\mathcal{A}$ *is a set of arrows and* $\mathcal{A} = \{(S, D) \mid S, D \in \mathcal{N}\}$.

The attribute-dependency relation of two nodes in the DG is defined as follows.

DEFINITION 2. Dependency Relation. *Given two nodes, A and B, in a DG* $= (\mathcal{N}, \mathcal{A})$*, i.e.* $A, B \in \mathcal{N}$*, A and B have an* attribute-dependency *relation or simply* dependency *relation, denoted as DR*(A, B)*, iff*: (1) $A = B$; (2) *there is an arrow from A to B, i.e.* $(A, B) \in \mathcal{A}$*; or* (3) *there is a node* $X \in \mathcal{N}$*, such that DR*(A, X) *and DR*(X, B).

From the graphic point of view, a pair of nodes, A and B, has a dependency relation if there is a sequence of arrows from node A to node B. Terms such as dependency relation, arrow, and connection mean the same thing and are used interchangeably in this paper. The notation $DR(A, B)^d$ is used to indicate the *distance d* (i.e. the number of arrows) between two dependent nodes A and B. A node can be regarded as self-dependent, i.e. $d = 0$. The dependency relations among nodes in Figure 2a are: $DR(A, A)^0$, $DR(B, B)^0$, $DR(C, C)^0$, $DR(A, B)^1$, $DR(B, C)^1$, and $DR(A, C)^2$.

For simplicity and clarity in discussing dependency conflict resolution, we assume: (1) each node has only one attribute (hence a node may have only one input arrow but multiple output arrows); and (2) a node always propagates the input attribute value as-is (i.e. without making any change to it). According to Definition 1 and 2, and above assumptions, a *valid* DG must meet three conditions: (1) there is no *dangling arrow* in the DG; (2) there is no more than one input arrow to each node; and (3) dependent nodes have the same attribute value. In real systems, however, each node may have multiple attributes (and thus multiple input arrows), and may apply an internal function to transform the input value (received from an upstream node) into a new output value (to be propagated to downstream nodes). The discussions and solutions presented in this paper are generalizable to systems which allow multiple attributes per node and functional transformation at each node.

Operations in a DG

G-objects in a graphic design system can be addressed in various ways, provided they are unique and consistent for the same replicas [1,15]. For clarity, g-objects in our examples are identified simply by using unique alphabets.

In attribute-independent graphic editing systems, there are three basic operations: a *Create* operation creates a new g-object, a *Delete* operation removes an existing g-object, and an *Update* operation changes the attribute value of a g-object [15]. DG-based graphic design systems also have these operations, but the *Update* operation has an additional effect of propagating the change of an attribute value to all dependent g-objects. DG-based graphic design systems have two additional basic operations: a *Connect* operation establishes a dependency relation of two g-objects, and propagates the attribute value of one g-object (the *source*) to another g-object (the *destination*), which may further propagates the value to downstream g-objects (if any); and a *Disconnect* operation removes the dependency relation of two g-objects. It is worth highlighting that a *Disconnect* cannot reverse the effects of a *Connect* because a *Disconnect* can only remove a dependency relationship established by a *Connect,* but cannot eliminate the attribute value propagated by a *Connect*. All operations except *Update* may change the structure of a DG; *Update* may only change attribute values of DG nodes.

A general requirement for a DG operation is that its execution must transform a valid DG state into another valid DG state. To ensure that the execution of a DG operation will lead to a valid DG state, certain specific conditions must be met for an operation to be generated from and/or executed on a document state. For example, a *Delete* operation must not target an object with any connection because otherwise the execution of the *Delete* operation will lead to an invalid DG state with dangling arrows. Such conditions are called operation *preconditions*. In single-user 3D design systems, operation preconditions are enforced by the underlying system: when a user is to generate an operation, the underlying system will check the current DG state to see whether the preconditions of the operation are met. If so, the operation will be generated and executed successfully; otherwise, the generation will fail and have no effect on the DG state. As we will explain later, operation preconditions play an important role in detecting and resolving dependency conflicts.

The execution effect of an operation on the DG state can be specified as a set of *postconditions*, which capture the changes to the DG state. One general postcondition for all operations is the resulting DG state must be valid. Postconditions can be used to evaluate preconditions for follow up operations and detecting potential conflicts as well. In the following, we give precise specifications of the five DG operations, together with their preconditions and postconditions, on the initial DG state $G = (\mathcal{N}, \mathcal{A})$.

1. ***X* := *CR*(*g*)**: *Create* a new g-object g and return the identifier X.
   a. ***Pre-CR***: none.
   b. ***Post-CR*:** a new g-object g with the identifier X is added into the DG, i.e. $G = (\mathcal{N} \cup \{X\}, \mathcal{A})$.
2. ***DL*(*X*)**: *Delete* an existing g-object X.
   a. ***Pre-DL***: X must exist in the DG and have no input/output connection, i.e. $(X \in \mathcal{N}) \wedge (\forall (S, D) \in \mathcal{A}\ (S \neq X \wedge D \neq X))$.
   b. ***Post-DL***: X is removed from the DG, i.e. $G = (\mathcal{N} \setminus \{X\}, \mathcal{A})$.
3. ***UP*(*X*, *v*)**: *Update* the attribute value of a g-object X with v.

   a. ***Pre-UP***: X must exist in the DG and have no input connection, i.e. $(X \in \mathcal{N}) \wedge (\forall (S, D) \in \mathcal{A} (D \neq X))$.
   b. ***Post-UP***: attribute values of X and its downstream g-objects are updated with value v, *i.e.* $G = (\mathcal{N}', \mathcal{A})$ where $\mathcal{N}' = \{Y \mid (Y \in \mathcal{N}) \wedge (DR(X, Y)^d \rightarrow Y.v = v)\}$.
4. ***CN(S, D)***: *Connect* the source g-object S to the destination g-object D.
   a. ***Pre-CN***: S and D must exist in the DG and D must not have any input connection, i.e. $(S, D \in \mathcal{N}) \wedge (\forall (X, Y) \in \mathcal{A} (Y \neq D))$.
   b. ***Post-CN***: a connection from S to D is established and the attribute value of S is propagated to D and its downstream g-objects, i.e. $G = (\mathcal{N}', \mathcal{A}')$, where $\mathcal{N}' = \{Y \mid (Y \in \mathcal{N}) \wedge (DR(S, Y)^d \rightarrow Y.v = S.v)\}$ and $\mathcal{A}' = \mathcal{A} \cup \{(S, D)\}$.
5. ***DC(S, D)***: *Disconnect* the source g-object S from the destination g-object D.
   a. ***Pre-DC***: S and D must exist in the DG and a connection between S and D must also exist, i.e. $(S, D \in \mathcal{N}) \wedge ((S, D) \in \mathcal{A})$.
   b. ***Post-DC***: the connection from S to D is removed, i.e. $G = (\mathcal{N}, \mathcal{A} \setminus \{(S, D)\})$.

The above operation notations capture only essential parameters for discussing dependency conflict resolution. In real systems, DG operations may carry additional parameters required by the application. In the absence of concurrency, a precondition met at the local site when an operation was generated can also be met when this operation is executed at a remote site. In the presence of concurrency in real-time collaboration, however, a precondition met at the local site may be violated at a remote site due to the execution of other concurrent operations. Due to different execution orders of concurrent operations, the same operation may succeed in one execution order but fail in another order, leading to inconsistent document states.

DEPENDENCY CONFLICT RELATION AND DETECTION

Conflict Definition

Operations in collaborative design systems may have dependency conflict/compatible relations as defined below.

DEFINITION 3. ***Dependency Conflict Relation "⊗".*** *Given two DG operations, O_1 and O_2, they have a* Dependency Conflict Relation*, denoted as $O_1 \otimes O_2$, iff*: (1) *O_1 and O_2 are defined on the same DG state*; (2) *targets of O_1 and O_2 have dependency relations*; *and* (3) *different execution orders of O_1 and O_2 result in different DG states, i.e. different DG structures and/or g-objects values.*

DEFINITION 4. ***Compatible Relation "⊙".*** *Given two operations, O_1 and O_2, they have a* Compatible Relation*, denoted as $O_1 \odot O_2$, iff they do not have a dependency conflict relation, i.e. $\neg(O_1 \otimes O_2)$.*

The overlapping conflict relation, as defined in [15], can be viewed as a special case of the dependency conflict relation

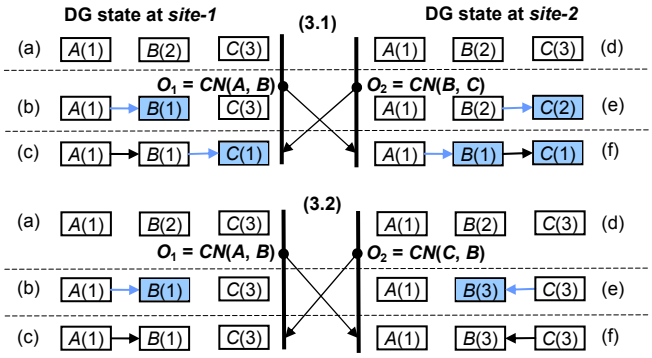

Figure 3. Concurrent operations with common targets: (3.1) compatible relation; and (3.2) conflict relation.

since the overlapping g-objects targeted by concurrent operations can be viewed as self-dependent g-objects with a zero distance dependency relation (i.e. $DR(A, B)^0 \leftrightarrow A = B$).

Target-dependency is a necessary but insufficient condition for two operations to have a dependency conflict relation. To illustrate, consider the scenario in Figure 3.1: two connect operations, $O_1 = CN(A, B)$ and $O_2 = CN(B, C)$, are generated concurrently by *site-1* and *site-2*, respectively, over an initial DG state: $A(1)$, $B(2)$, and $C(3)$ (Figure 3.1a and d). Regardless on which orders operations are executed, O_1 followed by O_2, or O_2 followed by O_1, the DG becomes $A(1) \rightarrow B(1) \rightarrow C(1)$ consistently after both operations have been executed as shown in Figure 3.1c and f. In this case, the two operations are compatible even though they share one common target g-object B (thus being dependent).

Consider a similar scenario in Figure 3.2 with the same initial DG state but slightly different parameters for O_2, where $O_2 = CN(C, B)$. Suppose these two operations are executed in the order of O_1 followed by O_2 (at *site-1*). After the execution of O_1, the DG becomes $A(1) \rightarrow B(1)$, $C(3)$ (Figure 3.2b). The execution of O_2 failed due to the *precondition* that the destination g-object cannot have any existing input connection as specified in *Pre-CN*, thus the DG cannot be changed by O_2 (Figure 3.2c). Now suppose these two operations are executed in the reverse order of O_2 followed by O_1 (at *site-2*). After the execution of O_2, the DG becomes $A(1)$, $B(3) \leftarrow C(3)$ (Figure 3.2e). The execution of O_1 also failed due to the violation of the same precondition *Pre-CN* and the DG cannot be changed by O_1 either (Figure 3.2f). After both operations are executed, the dependency relation and attribute value of B are clearly different on different execution orders. In this case, the two operations conflict due to a shared g-object B.

To summarize, in both Figure 3.1 and 3.2, the two operations share a common (thus dependent) target g-object B, but they conflict only in the case of Figure 3.2. The inconsistent result (thus conflict) was caused by the failure of one of the operations due to a precondition violation. Preconditions are useful hints for conflict detection, but precondition-violation alone is neither necessary nor sufficient for determining whether two operations conflict.

	$CR(g_2)$	$DL(X_2)$	$UP(X_2, v_2)$	$CN(S_2, D_2)$	$DC(S_2, D_2)$
$CR(g_1)$	⊙	⊙	⊙	⊙	⊙
$DL(X_1)$		⊙	⊙	⊗ ↔ $DR(X_1, S_2)^0$ or $DR(X_1, D_2)^0$	⊙
$UP(X_1, v_1)$			⊗ ↔ $DR(X_1, X_2)^0$	⊙	⊗ ↔ $DR(X_1, S_2)^d$
$CN(S_1, D_1)$				⊗ ↔ $DR(D_1, D_2)^0$	⊗ ↔ $DR(D_1, S_2)^d$
$DC(S_1, D_1)$					⊙

Table 1. Conflict relations and detection conditions

There are subtle scenarios, in which precondition-violating operations may not conflict, and precondition-satisfying operations may conflict. The next section provides precise and theoretically verified conflict detection conditions between every pair of the five DG operations.

Conflict/Compatible Relations

Based on comprehensive analysis, we summarize the conflict/compatible relations between all pairs of DG operations in Table 1. Some pairs of operations are always compatible because either their targets do not (or cannot) have any dependency relation (e.g. a *CR* with all other operations) or their executions in different orders always result in the same DG state (e.g. a pair of *DL*s). Some other pairs may be either compatible or conflicting, depending on the dependency relations of their targets and whether their executions in different orders result in the same or different DG states. There are five such pairs: *DL*-*CN*, *UP*-*UP*, *UP*-*DC*, *CN*-*CN*, and *CN*-*DC*, of which precise conflict conditions are filled in the corresponding cells of Table 1.

We have conducted theoretical verification on the conflict conditions for every pair of operations. Due to space limitation, only the proof for a pair of *UP* and *DC* is presented here to show the basic ideas and methods of our verification. First, we introduce Lemma 1 which establishes some necessary conditions for an *UP* to conflict with another operation generated from the same DG state.

***Lemma 1*:** Given an $UP(X_1, v)$ and any operation O with a target X_2 generated from the same DG state $G = (\mathcal{N}, \mathcal{A})$, they are conflicting only if X_1 is overlapping with or on the upstream of X_2, i.e. $DR(X_1, X_2)^d$, where $d \geq 0$.

***Proof*:** First, for an $UP(X_1, v)$ to conflict with O, their targets must have a dependency relation according to Definition 3. Second, it is impossible for their targets to have a dependency relation other than X_1 is overlapping with or on the upstream of X_2, i.e. $DR(X_1, X_2)^d$, where $d \geq 0$. Otherwise, X_1 would have an input connection, i.e. $\exists Y \in \mathcal{N}$ $((Y, X_1) \in \mathcal{A})$, which contradicts with the *Pre-UP* for generating $UP(X_1, v)$. Hence, the lemma follows.

***Theorem 1*:** An $UP(X, v)$ conflicts with a $DC(S, D)$ defined on the same DG state $G = (\mathcal{N}, \mathcal{A})$ *iff* X is overlapping or on the upstream of S, i.e. $UP(X, v) \otimes DC(S, D) \leftrightarrow DR(X, S)^d$.

***Proof*:** First, we show the theorem holds for $UP(X, v) \otimes DC(S, D) \longrightarrow DR(X, S)^d$. From the precondition that $UP(X, v)$ conflicts with $DC(S, D)$, we derive X is overlapping or on the upstream of S, i.e. $DR(X, S)^d$, where $d \geq 0$, by Lemma 1. Second, we show the theorem holds for $DR(X, S)^d \longrightarrow UP(X, v) \otimes DC(S, D)$. In case $UP(X, v)$ is executed before $DC(S, D)$: (1) $UP(X, v)$ will update X and its downstream with value v and result in $G_1 = (\mathcal{N}_1, \mathcal{A}_1)$ where $\mathcal{N}_1 = \{Y \mid (Y \in \mathcal{N}) \wedge (DR(X, Y)^d \rightarrow Y.v = v)\}$ and $\mathcal{A}_1 = \mathcal{A}$; and (2) $DC(S, D)$ will only remove the arrow (S, D) from $\mathcal{A}$ and result in $G_1' = (\mathcal{N}_1', \mathcal{A}_1')$ where $\mathcal{A}_1' = \mathcal{A} \setminus \{(S, D)\}$ and $\mathcal{N}_1' = \{Y \mid (Y \in \mathcal{N}) \wedge ((DR(X, Y)^d \vee DR(D, Y)^d) \rightarrow Y.v = v)\}$ because $DC(S, D)$ has split X's downstream nodes (in G_1), which share the same attribute value as X, into two groups: (a) nodes from X to S (due to $DR(X, S)^d$), expressed by $DR(X, Y)^d$, and (b) nodes from D and downwards (due to the connection (S, D)), expressed by $DR(D, Y)^d$. In case $DC(S, D)$ is executed before $UP(X, v)$: (1) $DC(S, D)$ will remove the arrow (S, D) from $\mathcal{A}$ and result in $G_2 = (\mathcal{N}_2, \mathcal{A}_2)$ where $\mathcal{N}_2 = \mathcal{N}$ and $\mathcal{A}_2 = \mathcal{A} \setminus \{(S, D)\}$; and (2) $UP(X, v)$ will update X and its downstream nodes, up till S due to $DR(X, S)^d$, and result in $G_2' = (\mathcal{N}_2', \mathcal{A}_2)$, where $\mathcal{N}_2' = \{Y \mid (Y \in \mathcal{N}) \wedge (DR(X, Y)^d \rightarrow Y.v = v)\}$. From the fact that $(DR(D, Y)^d \rightarrow Y.v = v)$ is in G_1' but not G_2', we have $G_1' \neq G_2'$ and conclude $UP(X, v)$ and $DC(S, D)$ conflict under the condition $DR(X, S)^d$ by Definition 3. The theorem holds under all circumstances.

COMBINED EFFECTS OF CONCURRENT OPERATIONS

General Requirements on Combined Effects

The following three general requirements are imposed on the combined effects of concurrent operations, with or without dependency conflicts:

1. ***Convergence*:** combined effects must be identical at all sites, in terms of attribute values of all g-objects and their dependency relations in the DG.
2. ***Intention-confined effect*:** combined effects should be confined by user-generated operations; namely, no changes can be made to dependency relations or attribute values of g-objects in the DG, unless such changes are initiated by user-generated operations.
3. ***Aggressive effect preservation*:** combined effects should preserve effects of compatible operations fully, and effects of conflicting operations as much as possible.

These requirements are not a replacement for the general consistency requirements for collaborative editors in [19]. The first requirement is a specialization of the *convergence property* in [19]: covering not only the value part of g-objects but also the structure of the DG. The second and third requirements together define a special form of the *intention preservation* property in [19]: changes made to the DG must be initiated by user-generated operations and as much operation effects as possible should be preserved even in the face of conflict, which is desirable in real-time collaborative working environments with optimistic concurrency control strategies [4,6,15,20]. The *causality preservation* property in [19] should also be preserved in collaborative 3D design systems, but is not listed here as it is irrelevant to defining combined effects of operations.

Combined Effects for DG Operations

Guided by the general requirements on combined effects, we define six concrete combined effects for operations in Table 1 as follows.

1. ***CE1***: the combined effect for compatible operations is the union of the effects of all operations, which can be achieved by executing compatible operations in any causality-preserving order.
2. ***CE2*:** the combined effect for conflicting *DL* and *CN*, i.e. $DL(X) \otimes CN(S, D)$, is: (1) the target X is deleted, which preserves the effect of $DL(X)$; and (2) there exists no connection from S to D (as one of them must overlap with X and has been deleted) but the attribute value of S has been propagated to D (if it is not the target of $DL(X)$), which preserves part of the effect of $CN(S, D)$.
3. ***CE3*:** the combined effect for conflicting *UP*s, i.e. $UP(X, v_1) \otimes UP(X, v_2)$, is the attribute value of the overlapping target X and its downstream are updated with either v_1 or v_2, depending on the priority between the two *UPs*. This is a case that a conflict is resolved by priority and only one operation effect is preserved.
4. ***CE4*:** the combined effect for conflicting *UP* and *DC*, i.e. $UP(X, v) \otimes DC(S, D)$, is: (1) the attribute value of X is updated to v and propagated to its downstream g-objects (including S and D), which preserves the full effect of $UP(X, v)$; and (2) the connection from S to D is broken, which preserves the effect of $DC(S, D)$. This is a case that all conflicting operations effects are preserved.
5. ***CE5*:** the combined effect for conflicting *CN*s, i.e. $CN(S_1, D) \otimes CN(S_2, D)$, is: only one connection is successful, depending on the priority of the two *CN*s. This is another case that a conflict is resolved by priority and only one operation effect is preserved.
6. ***CE6*:** the combined effect for conflicting *CN* and *DC*, i.e. $CN(S_1, D_1) \otimes DC(S_2, D_2)$, is: (1) a connection from S_1 to D_1 is established, and the attribute value of S_1 is propagated to its downstream g-objects (including D_1, S_2 and D_2), which preserves the full effect of $CN(S_1, D_1)$; and (2) the connection from S_2 to D_2 is removed, which preserves the effect of $DC(S_2, D_2)$. This is another case that all conflicting operations effects are preserved.

Various priority schemes can be used in the priority-based resolution for $UP \otimes UP$ and $CN \otimes CN$ cases. For example, priorities can be based on a total ordering of operations [17]. To facilitate total ordering and concurrency checking, operations are time-stamped with state vectors [17].

OT-BASED DEPENDENCY CONFLICT RESOLUTION

Basic Idea

From the point of view of consistency maintenance, the basic idea of OT is to transform an operation defined on one context (i.e. a document state) into another operation defined on a new context, so that the transformed operation can be correctly executed on the new context and achieve document consistency in the face of concurrent operations [16,17]. This basic OT idea can be applied for consistency maintenance in the face of concurrent operations with dependency conflict relations.

To illustrate this, consider again the scenario in Figure 2, in which operations are always executed in their original forms and different execution orders of the two dependency conflicting operations O_1 and O_2 result in inconsistent DG states. This problem can be resolved if a remote operation is transformed against the other concurrent operation in such a way that the transformed operation can achieve the same and correct combined effect. For the scenario in Figure 2, the combined effect for a pair of *UP* and *DC* operations should be the *CE4*, which preserves effects of all operations. We observed that the effects of O_1, when generated at its local *site-1* (Figure 2b), are to update the attribute value of not only g-object A, but also the two dependent g-objects: B and C. However, these effects cannot be fully achieved if O_1 is executed (as-is) on a different DG states at *site-2*, where C is no longer depending on A (the target of O_1) due to the prior execution of O_2 (Figure 2e). To achieve the full effects of O_1 on the new DG state, an additional *UP* operation should be performed on C. From the OT's point of view, O_1 should be transformed (against O_2) into a list of two operations: one is the original *UP* operation on g-object A and the other is a new *UP* operation on g-object C. The execution of the transformed O_1 would achieve the full effects of O_1. At *site-1*, O_2 should be transformed (against O_1) as well; but this transformation does not change O_2. In this way, a consistent and correct combined effect *CE4* can be achieved at both sites: the candle and its holder are resized and fit into the resized box. This combine effect is also a desirable one because the user at *site-2* can directly rescale only the width of the box to achieve the objective of fitting multiple candles in the box.

Transformation Control Algorithm

An OT system consists of generic control algorithms and domain-specific transformation functions [16,17]. OT control algorithms determine the correct order and conditions for operation transformations and executions, while transformation functions perform the actual transformation of an operation against another operation according to the types and parameters of the operations. One important condition is that for two operations to be transformed with each other, they must be defined on the same document state (i.e. *context-equivalent*) [12,18]. The violation of this condition has been well known as the root problem of some prior OT algorithms.

Past research of OT has devised various transformation control algorithms that preserve the context-equivalency condition, e.g. adOPTed [14], GOT/GOTO [16,17], Jupiter [12], COT [18], etc. These control algorithms are independent of application domains and can be directly adopted for supporting dependency conflict resolution. We designed our transformation functions based on two requirements imposed by control algorithms: (1) the transformation functions are *Inclusion Transformation* (IT),

which is to include the impact of the reference operation [16,17]; and (2) the transformation functions preserve the *Convergence Property 1* (CP1), i.e. applying O_x followed by $O_y' = T(O_y, O_x)$ on a DG state G achieves the same effect as applying O_y followed by $O_x' = T(O_x, O_y)$ on the same G [16]. COT and Jupiter are example of control algorithms that impose only these requirements and any of them can be adapted (i.e. to call the *LOT* function defined in the next sub-section, instead of the *T* function) and integrated with the dependency conflict resolution presented in this paper.

Transformation of Lists of Operations

OT-based dependency conflict resolution may transform an operation into a *list of operations* (*LO*) in resolving dependency conflict, as seen in the previous example. To facilitate transformations at the *LO* level, i.e. transforming one LO_1 against another context-equivalent LO_2, a list OT function *LOT* is defined in the pseudocode form as follows.

Algorithm-1: *LOT*(LO_1, LO_2)

```
1:  var LO1' := [ ];
2:  Repeat until LO1 = [ ]:
3:      if LO2 = [ ] then { LO1 := LO1' + LO1; return; }
4:      else{
5:          Remove Ox from LO1, where Ox is the first element in LO1;
6:          Remove Oy from LO2, where Oy is the first element in LO2;
7:          Ox := T(Ox, Oy); LOy := T(Oy, Ox);
8:          LOT(LOx, LO2);
9:          LO1' := LO1' + LOx;  LO2 := LOy + LO2;  }
10: LO1 := LO1';
```

In *LOT*, the following repeats until LO_1 becomes empty. If LO_2 is empty, it inserts LO_1', which is the transformation result so far, in front of LO_1 and returns (1.3); otherwise, it removes the first element O_x from LO_1 and O_y from LO_2 (1.5, 1.6). Then, it transforms O_x against O_y and O_y against O_x by the transformation function $T(O_x, O_y)$ (see the next section for *T* function definition), which return LO_x and LO_y, respectively (1.7). Afterwards, the *LOT* function is recursively invoked to further transform LO_x against the remaining operations in LO_2 (1.8). After the recursive call, LO_x stores the transformation result of O_x against all operations in the initial LO_2, and LO_2 stores the transformation result of all operations but O_y in the initial LO_2 against O_x. Then, LO_x is merged with LO_1' (i.e. $LO_1' = LO_1' + LO_x$) and LO_y, which is the transformed value of the first element of the initial LO_2, is inserted in front of the transformed LO_2, i.e. ($LO_2 = LO_y + LO_2$) (1.9). Finally, after all operation in LO_1 has been transformed (i.e. when LO_1 becomes empty), LO_1' overwrites LO_1 (1.10), so LO_1 contains the transformed operations against LO_2.

There are two things that we should point out. First, the *LOT* function not only transforms LO_1 against LO_2, but also transforms LO_2 against LO_1, which is necessary to ensure the context-equivalency of later transformations. Second, for the concurrency/context checking, all operations in *LO* share the same state/context vector [17,18].

Transformation Functions

Algorithm-2: $T(O_x, O_y)$

```
1: if type(Ox) = DL and type(Oy) = CN then
2:     LO := Tdlcn(Ox, Oy);
3: else if type(Ox) = CN and type(Oy) = DL then
4:     LO := Tcndl(Ox, Oy);
5: else if ... // other cases are omitted for conciseness
6: return LO;
```

The transformation function $T(O_x, O_y)$ in Algorithm-2 transforms O_x against O_y in such a way that the effect of O_y is included in O_x (i.e. *IT* [16,17]). The transformation function has the following precondition and postcondition:

- ***Pre-T***: O_x and O_y are context-equivalent.
- ***Post-T***: an operation O is included in the returned *LO* only if its precondition is satisfied after sequentially executing other preceding operations in *LO*; and the execution of O_y followed by all operations in *LO* achieves the combined effects defined in *CE1–CE6*.

The postcondition also implies that any operation with a precondition violation would not be included in *LO* and has no impact on subsequent executions and transformations. This is important for ensuring the correctness of the transformation.

The following basic strategies are used in designing the transformation function: assume O_y has been executed, assess the impact of O_y on the DG state and derive what should be executed to achieve the combined effects of O_x and O_y according to *CE1–CE6*. Based on the types of O_x and O_y, function $Tpq(O_x, O_y)$ is invoked to transform O_x to achieve a suitable combined effect. In each $Tpq(O_x, O_y)$: (1) if $O_x \otimes O_y$ (see Table 1 for conflict detection conditions), O_x is transformed into a list of operations that can jointly achieve the suitable combined effect according to *CE2–CE6*; (2) if $O_x \odot O_y$, then O_x is transformed into $[O_x]$ in case that its preconditions are met, or [ ] in case that its preconditions are violated (so O_x should not have any effect in later executions or transformations), according to *CE1*.

According to Table 1, conflicts may occur only for five pairs of operation types. For each pair of operation types, *p* and *q*, two transformation functions *Tpq* and *Tqp* need to be defined for handling two different orders of execution. Since pairs *UP-UP* and *CN-CN* need only one function for each case, a total of *eight* transformation functions cover all conflict resolution cases. For the compatible pairs, there are four cases that involve precondition violations and require the operation to be nullified so that it will not affect subsequent transformations. The pseudocode of the transformation functions are defined in Algorithm-3.

Algorithm-3: Transformation Functions

***Tdlcn*(*DL*(*X*), *CN*(*S*, *D*)):**

```
1: if DL(X) ⊗ CN(S, D) then return [DC(S, D), DL(X)];
2: else return [DL(X)]; // for DL(X) ⊙ CN(S, D)
```

***Tcndl*(*CN*(*S*, *D*), *DL*(*X*)):**

```
1: if DL(X) ⊗ CN(S, D) then {
2:     if S ≠ X then return [ ];  // i.e. D = X
```

3: **else return** [$UP(D, S.v)$]; }
4: **else return** [$CN(S, D)$]; // for $CN(S, D) \odot DL(X)$

***Tupup*($UP_1(X_1, v_1)$, $UP_2(X_2, v_2)$):**
1: **if** $UP_1(X_1, v_1) \otimes UP_2(X_2, v_2)$ **then** {
2: **if** *HigherPriority*(UP_1, UP_2) **then return** [$UP_1(X_1, v_1)$];
3: **else return** [ ]; }
4: **else return** [$UP_1(X_1, v_1)$]; // for $UP_1(X_1, v_1) \odot UP_2(X_2, v_2)$

***Tupdc*($UP(X, v)$, $DC(S, D)$):**
1: **if** $UP(X, v) \otimes DC(S, D)$ **then return** [$UP(X, v)$, $UP(D, v)$];
2: **else return** [$UP(X, v)$]; // for $UP(X, v) \odot DC(S, D)$

***Tdcup*($DC(S, D)$, $UP(X, v)$):**
return [$DC(S, D)$]; //for both $\otimes$ and $\odot$ relations

***Tcncn*($CN_1(S_1, D_1)$, $CN_2(S_2, D_2)$):**
1: **if** $CN_1(S_1, D_1) \otimes CN_2(S_2, D_2)$ **then** {
2: **if** *HigherPriority*(CN_1, CN_2) **then**
3: **return** [$DC(S_2, D_2)$, $CN_1(S_1, D_1)$];
4: **else return** [ ]; }
5: **else return** [$CN_1(S_1, D_1)$]; // for $CN_1(S_1, D_1) \odot CN_2(S_2, D_2)$

***Tcndc*($CN(S_1, D_1)$, $DC(S_2, D_2)$):**
1: **if** $CN(S_1, D_1) \otimes DC(S_2, D_2)$ **then**
2: **return** [$CN(S_1, D_1)$, $UP(D_2, S_1.v)$];
3: **else return** [$CN(S_1, D_1)$]; // for $CN(S_1, D_1) \odot DC(S_2, D_2)$

***Tdccn*($DC(S_1, D_1)$, $CN(S_2, D_2)$):**
return [$DC(S_1, D_1)$]; //for both $\otimes$ and $\odot$ relations

***Tdldl*($DL_1(X_1)$, $DL_2(X_2)$):**
1: **if** $X_1 = X_2$ **then return** [ ]; //*Pre-DL* violation
2: **else return** [$DL_1(X_1)$];

***Tupdl*($UP(X_1, v)$, $DL(X_2)$):**
1: **if** $X_1 = X_2$ **then return** [ ]; //*Pre-UP* violation
2: **else return** [$UP(X_1, v)$];

***Tupcn*($UP(X, v)$, $CN(S, D)$):**
1: **if** $X = D$ **then return** [ ]; // *Pre-UP* violation
2: **else return** [$UP(X, v)$];

***Tdcdc*($DC_1(S_1, D_1)$, $DC_2(S_2, D_2)$):**
1: **if**($S_1 = S_2$ **and** $D_1 = D_2$) **then return** [ ]; //*Pre-DC* violation
2: **else return** [$DC_1(S_1, D_1)$];

//Details for transformation functions *Tcrcr*, *Tcrdl*, *Tdlcr*, *Tcrup*, //*Tupcr*, *Tcrcn*, *Tcncr*, *Tcrdc*, *Tdccr*, *Tdlup*, *Tdldc*, *Tdcdl*, and *Tcnup* //are omitted because they handle pairs of compatible operations //whose pre-conditions are always satisfiable in execution, and hence //always return the original operation.

The combined effect for a pair of concurrent $DL(X)$ and $CN(S, D)$ should be *CE2* in both execution orders. (1) The *Tdlcn* function handles the case that $DL(X)$ is executed after $CN(S, D)$. If conflict is detected, $DL(X)$ is transformed into a list of two operations: $DC(S, D)$ to remove the connection from S to D for satisfying the *Pre-DL*, and then $DL(X)$ to delete the target X. Otherwise (compatible), it returns [$DL(X)$]. (2) The *Tcndl* function handles the case that $CN(S, D)$ is executed after $DL(X)$. In case of conflict, it returns an empty list [ ] if $S \neq X$; else it returns an $UP(D, S.v)$ to propagate the value of S (i.e. $S.v$) to D. Otherwise (compatible), it returns [$CN(S, D)$].

The combined effect for a pair of concurrent *UP* is *CE3*. Only one *Tupup* function is needed. In case of conflict, it returns [$UP_1(X_1, v_1)$] if $UP_1(X_1, v_1)$ has a higher priority than $UP_2(X_2, v_2)$; else it returns [ ]. In case of compatible, it returns [$UP_1(X_1, v_1)$]. The combined effect for a pair of concurrent $UP(X, v)$ and $DC(S, D)$ is *CE4*. (1) The *Tupdc* function handles the case that $UP(X, v)$ is executed after $DC(S, D)$. If conflict, it returns a list of two operations: the original $UP(X, v)$, followed by a new $UP(D, v)$ to propagate the attribute value v to D. If compatible, it returns [$UP(X_1, v)$]. (2) The *Tdcup* function handles the case that $DC(S, D)$ is executed after $UP(X, v)$. It returns [$DC(S, D)$] regardless whether the two operations conflict or not.

The combined effect for a pair of concurrent *CN* operations is *CE5*. Only one *Tcncn* function is needed. In case of conflict, if $CN_1(S_1, D_1)$ has a higher priority than $CN_2(S_2, D_2)$, it returns a list of two operations: $DC(S_2, D_2)$ to break the connection from S_2 to D_2, followed by $CN_1(S_1, D_1)$; else it returns [ ]. In case of compatible, it returns [$CN_1(S_1, D_1)$]. The combined effect for a pair of concurrent $CN(S_1, D_1)$ and $DC(S_2, D_2)$ is *CE6*. (1) The *Tcndc* function handles the case that $CN(S_1, D_1)$ is executed after $DC(S_2, D_2)$. If conflict, it returns a list of two operations: $CN(S_1, D_1)$, followed by an $UP(D_2, S_1.v)$ to propagate the attribute value of S_1 to D_2 and its downstream. If compatible, it returns [$CN(S_1, D_1)$]. (2) The *Tdccn* function handles the case that $DC(S_2, D_2)$ is executed after $CN(S_1, D_1)$. It returns [$DC(S_2, D_2)$] regardless whether the two operations conflict or not.

The combined effect for other pairs of operations is *CE1* as they are compatible pairs. However, based on *Post-T*, operations with precondition violations must be nullified. The *Tdldl*, *Tupdl*, *Tupcn*, and *Tdcdc* functions handle the cases where two operations are always compatible but the preconditions may be violated under certain conditions: if preconditions are violated, these functions return an empty list; otherwise, they return the original operation. It should be pointed out that in *Tdlcn* and *Tcncn*, the *DC* is not to undo the conflicting operation *CN* (thus move *backward* to an old *DG* state) as in some serialization approaches like [8], but to move *forward* to a new *DG* state that meets the precondition for the *DL* and *CN*. In fact, undoing *CN* cannot be achieved by simply removing the connection because it requires eliminating the value propagation effect as well.

Convergence property of Transformation Functions

For any pair of DG operations O_x and O_y, their combined effect is defined by *CE1–CE6*, regardless of their execution orders. Let G be a *DG* state, $O_x' = T(O_x, O_y)$ and $O_y' = T(O_y, O_x)$. The T function is required to preserve the following *Convergence Property*:

$$G \circ O_x \circ O_y' = G \circ O_y \circ O_x',$$

which means applying O_x followed by O_y' on a state G achieves the same effect as applying O_y followed by O_x' on the same G (i.e. TP in [4], TP1 in [16], and CP1 in [18]).

We have conducted theoretical verification on the convergence property for every pair of DG operations. Due to space limitations, we only include the proof of one pair, $UP(X, v)$ and $DC(S, D)$, to show the basic ideas that can be applied to the proof of other pairs of operations.

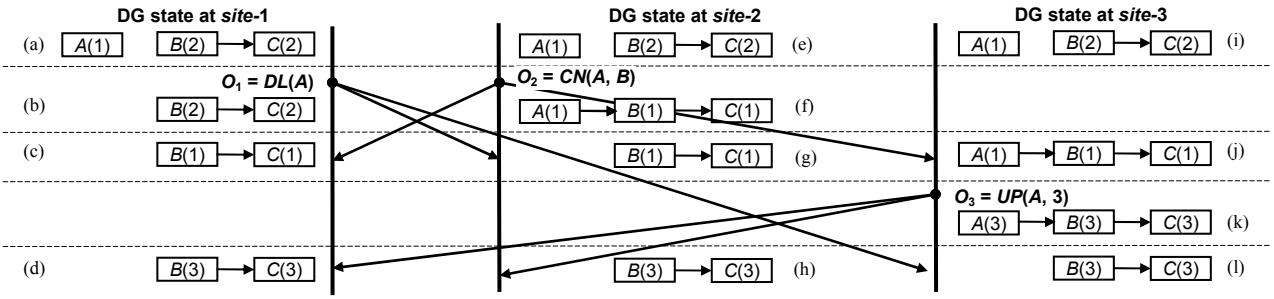

Figure 4. An example of dependency conflict resolution.

Theorem 2: Given two operations, $O_x = UP(X, v)$ and $O_y = DC(S, D)$, defined on the same state $G = (\mathcal{N}, \mathcal{A})$, the convergence property $G \circ O_x \circ O_y' = G \circ O_y \circ O_x'$ is preserved for $O_x' = Tupdc(O_x, O_y)$ and $O_y' = Tdcup(O_y, O_x)$.

Proof: It is trivial (so details are omitted) to show convergence is preserved when O_x and O_y are compatible, i.e. $O_x \odot O_y$, since their preconditions are always satisfied in execution according to *Pre-UP*, *Pre-DC*, and Theorem 1. So, we only show convergence is preserved when they are conflicting, i.e. $O_x \otimes O_y$. From Theorem 1 and $O_x \otimes O_y$, we derive $DR(X, S)^d$. According to the definitions of *Tupdc* and *Tdcup*, we have $O_x' = [UP(X, v), UP(D, v)]$, and $O_y' = [DC(S, D)]$, respectively. In the following, we show that the states produced by executing O_x and O_y in different orders are the same under the condition $DR(X, S)^d$.

For the execution sequence $G \circ O_x \circ O_y'$: (1) O_x will update X and its downstream nodes, and result in $G_1 = (\mathcal{N}_1, \mathcal{A}_1)$, where $\mathcal{N}_1 = \{Y \mid (Y \in \mathcal{N}) \wedge (DR(X, Y)^d \rightarrow Y.v = v)\}$ (by *Post-UP*), and $\mathcal{A}_1 = \mathcal{A}$; (2) *Pre-DC* for O_y' must be met as (S, D) is in $\mathcal{A}_1$ and hence in $\mathcal{A}$, so O_y' will only remove the arrow (S, D) from $\mathcal{A}$ and result in $G_1' = (\mathcal{N}_1', \mathcal{A}_1')$, where $\mathcal{A}_1' = \mathcal{A} \setminus \{(S, D)\}$ (by *Post-DC*), and $\mathcal{N}_1' = \{Y \mid (Y \in \mathcal{N}) \wedge ((DR(X, Y)^d \vee DR(D, Y)^d) \rightarrow Y.v = v)\}$ because $DC(S, D)$ has split X's downstream nodes (in G_1), which share the same attribute value as X, into two groups: (a) nodes from X to S (due to $DR(X, S)^d$), expressed by $DR(X, Y)^d$, and (b) nodes from D and downwards (due to (S, D)), expressed by $DR(D, Y)^d$.

For the execution sequence $G \circ O_y \circ O_x'$: (1) O_y will remove the arrow (S, D) from $\mathcal{A}$ and result in $G_2 = (\mathcal{N}_2, \mathcal{A}_2)$, where $\mathcal{N}_2 = \mathcal{N}$ and $\mathcal{A}_2 = \mathcal{A} \setminus \{(S, D)\}$ (by *Post-DC*); and (2) O_x' contains two sub operations and their preconditions (*Pre-UP*) must be met since O_y removes but not add an arrow from/into the *DG*: (a) $UP(X, v)$ will update X and its downstream nodes, up till S due to $DR(X, S)^d$, and result in $G_2' = (\mathcal{N}_2', \mathcal{A}_2)$, where $\mathcal{N}_2' = \{Y \mid (Y \in \mathcal{N}) \wedge (DR(X, Y)^d \rightarrow Y.v = v)\}$ (by *Post-UP*); (b) $UP(D, v)$ will update D and its downstream nodes, expressed as $DR(D, Y)^d$, and results in $G_2'' = (\mathcal{N}_2'', \mathcal{A}_2)$, where $\mathcal{N}_2'' = \{Y \mid (Y \in \mathcal{N}) \wedge ((DR(X, Y)^d \vee DR(D, Y)^d) \rightarrow Y.v = v)\}$ (by *Post-UP*).

By comparing G_1' and G_2'', we have $G_1' = G_2''$, so the theorem holds when $O_x \otimes O_y$. Therefore, the theorem holds under all circumstances.

AN INTEGRATED EXAMPLE

The OT-based dependency conflict resolution presented in this paper has been used in the CoMaya real-time collaborative 3D design system [1]. In this section, we use the *classic dOPT-puzzle* scenario [16], as shown in Figure 4, to illustrate how the whole system works. Three operations, $O_1 = DL(A)$, $O_2 = CN(A, B)$, and $O_3 = UP(A, 3)$, are generated by *site-1*, *site-2*, and *site-3*, respectively, with the causal relations $O_1 \parallel (O_2 \rightarrow O_3)$. The initial DG states at all sites are: $A(1)$ and $B(2) \rightarrow C(2)$ (Figure 4a, e, and i). The processing of each operation at all three sites is as follows.

Site-1 in the order of O_1, O_2, and O_3 (Figure 4b – d):
1. O_1 is executed as-is and results in $B(2) \rightarrow C(2)$.
2. When O_2 arrives, $LOT([O_2], [O_1])$ is invoked and results in $[O_{2a}]$ where $O_{2a} = UP(B, 1)$, by *Tcndl* due to $O_2 \otimes O_1$. The execution of O_{2a} results in $B(1) \rightarrow C(1)$.
3. When O_3 arrives: (a) $LOT([O_1], [O_2])$ is first invoked to bring $[O_1]$ to the same context as O_3 (the key step to resolving the *dOPT-puzzle* [16]), which results in $[O_{1a}, O_1]$, where $O_{1a} = DC(A, B)$, by *Tdlcn* due to $O_1 \otimes O_2$; and (b) $LOT([O_3], [O_{1a}, O_1])$ is then invoked, which first results in $[O_3, O_{3a}]$, where $O_{3a} = UP(B, 3)$, by *Tupdc* due to $O_3 \otimes O_{1a}$, and then these results are transformed with $[O_1]$ into $[O_{3a}]$ by *Tupdl* due to $O_3 \odot O_1$, $O_{3a} \odot O_1$, and *Pre-UP* violation for O_3. The execution of O_{3a} results in $B(3) \rightarrow C(3)$.

Site-2 in the order of O_2, O_1, and O_3 (Figure 4f – h):
1. O_2 is executed as-is and results in $A(1) \rightarrow B(1) \rightarrow C(1)$.
2. When O_1 arrives, $LOT([O_1], [O_2])$ is invoked and results in $[O_{1a}, O_1]$ (see step 3(a) at *site-1*). The execution of O_{1a} followed by O_1 results in $B(1) \rightarrow C(1)$.
3. When O_3 arrives, it is processed in the same manner as in step 3 at *site-1* and results in $[O_{3a}]$. The execution of O_{3a} results in $B(3) \rightarrow C(3)$.

Site-3 in the order of O_2, O_3, and O_1 (Figure 4j – l):
1. O_2 is executed as-is and results in $A(1) \rightarrow B(1) \rightarrow C(1)$.
2. O_3 is executed as-is and results in $A(3) \rightarrow B(3) \rightarrow C(3)$.
3. When O_1 arrives: (a) $LOT([O_1], [O_2])$ is first invoked and results in $[O_{1a}, O_1]$ (see step 3(a) at *site-1*); and (b) $LOT([O_{1a}, O_1], [O_3])$ is then invoked and results in $[O_{1a}, O_1]$ due to $O_{1a} \odot O_3$ and $O_1 \odot O_3$. The execution of O_{1a} followed by O_1 results in $B(3) \rightarrow C(3)$.

The final DG states at all sites are the same: $B(3) \rightarrow C(3)$, as shown in Figure 4d, h, and l.

CONCLUSIONS AND FUTURE WORK

In this paper, we have contributed a novel OT technique to resolve dependency conflicts in DG-based real-time unconstrained collaborative 3D design systems. This work is the first to extend OT capability from *address*-dependency to *attribute*-dependency conflict resolution, and OT application scope from 1D/2D to 3D applications, thus advancing the state-of-the-art of OT in both theory and practical application.

Main elements of the contribution include: (1) formal definitions of attribute-dependency conflict relations, detection conditions, pre- and post-conditions for five generic primitive operations – *Create, Delete, Connect, Disconnect, and Update* – on a general DG model; (2) definitions of both general and operation-specific requirements on combined effects of concurrent operations that ensure convergence and preserve operation effects in the face of conflicts; (3) OT transformation techniques and functions for resolving attribute dependency conflicts and achieving required effects; and (4) theoretical verification of conflict conditions and the correctness of the proposed solution. The proposed solution has been implemented in the CoMaya collaborative 3D design system.

The OT-based dependency conflict resolution technique in this paper is designed for supporting five primitive graphic operations on a general DG model. Like other OT solutions based on primitive operations (e.g. character insert/delete), combined effects at the primitive operation level may not always meet the user needs when multiple primitive operations are considered together. In real world collaborative systems (e.g. CoMaya), user interactions may generate *compound* operations, each of which is composed of multiple primitive operations. Additional conflict resolution mechanisms are required to support complex semantics of compound operations in the face of conflict. We are in the process of extending the dependency conflict resolution technique from primitive to complex compound operations and incorporating creative conflict resolution in collaborative design process [19]. Another future work is to support group undo in the face of dependency conflicts.

ACKNOWLEDGEMENT

This work is partially carried out at BeingThere Centre sponsored by the Singapore NRF and Interactive & Digital Media Program Office, MDA; and partially supported by the Agency of Science, Technology and Research of Singapore under SERC Grant 0821010018. The authors wish to thank anonymous reviewers for their insightful and constructive comments and suggestions.

REFERENCES

1. Agustina, F. Liu, S. Xia, H.F. Shen, and C. Sun. CoMaya: incorporating advance collaboration capabilities into 3D digital media design tools. *Proc. ACM CSCW 2008*, 5–8.
2. J. Begole, R. Smith, C. Struble, and C. Shaffer. Flexible collaboration transparency: supporting worker independence in replicated application-sharing systems. *TOCHI*, 6(2), 1999, 95–32.
3. P. Dourish. Consistency guarantees: exploiting application semantics for consistency management in a collaboration toolkit. *Proc. ACM CSCW 1996*, 268–277.
4. C.A. Ellis and S.J. Gibbs. Concurrency control in groupware systems. In *ACM SIGMOD Record*, 18(2), 1989, 399–407.
5. R. Galli and Y. Luo. Mu3D: a causal consistency protocol for a collaborative VRML editor. *Proc. ACM Web3D-VRML 2000*, 53–62.
6. S. Greenberg and D. Marwood. Real time groupware as distributed system: concurrency control and its effect on the interface. *Proc. ACM CSCW 1994*, 207–217.
7. G. Hesina, D. Schmalstieg, A. Fuhrmann, and W. Purgathofer. Distributed Open Inventor: a practical approach to distributed 3D graphics. *Proc. ACM VRST 1999*, 74–81.
8. C. Ignat and M. Norrie. Draw-Together: graphical editor for collaborative drawing. *Proc. ACM CSCW 2006*, 269–278.
9. R. Li, and D. Li. Transparent sharing and interoperation of heterogeneous single-user applications. *Proc. ACM CSCW 2002*, 246–255.
10. J. Munson and P. Dewan. A concurrency control framework for collaborative systems. *Proc. ACM CSCW 1996*, 278–287.
11. M. Naef, E. Lamboray, O. Staadt, and M. Gross. The blue-c distributed scene graph. In *Proc ACM. IPT/EGCE Workshop 2003*, 125–133.
12. D.A. Nicholas, P. Curtis, M. Dixon, and J. Lamping. High-latency, low-bandwidth windowing in the Jupiter collaboration system. *Proc. ACM UIST 1995*, 111–120.
13. K.P. Picott, B. McPherson, W.D. Angus, and I.V. Nagendra. System and method for using dependency graphs for the control of a graphics creation process. United State Patent No. 5,808,625, 1998.
14. M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhauser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. *ACM CSCW 1996*,288–297.
15. C. Sun and D. Chen. Consistency maintenance in real-time collaborative graphics editing systems. *TOCHI*, 9(1), 2002, 1–41.
16. C. Sun and C. A. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. *Proc. ACM CSCW 1998*, 59–68.
17. C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and intention preservations in real-time cooperative editing systems. *TOCHI*, 5(1), 1998, 63–108.
18. D. Sun and C. Sun. Context-based operational transformation in distributed collaborative editing systems. *TPDS*, 20(10), 2009, 1454–1470.
19. D. Sun, C. Sun, S. Xia and HF Shen. Creative conflict resolution in collaborative editing systems. *Proc ACM CSCW 2012.*
20. C. Sun, H. Wen and H. Fan. Operational transformation for orthogonal conflict resolution in collaborative two-dimensional document editors. *Proc. ACM CSCW 2012*.
21. C. Sun, S. Xia, D. Sun, D. Chen, H.F. Shen, and W. Cai, Transparent adaptation of single-user applications for multi-user real-time collaboration. *TOCHI*, 13(4), 2006, 531–582.
22. H. Tramberend. Avocado: a distributed virtual reality framework. *Proc. IEEE VR 1999*, 14–21.