Maintaining Multi-way Dataflow Constraints in Collaborative Systems

Kai Lin, David Chen and Geoff Dromey
School of Information and Communication Technology, Griffith University
Brisbane, QLD 4111, Australia
Kai.Lin@student.griffith.edu.au,
{D.Chen, G.Dromey}@griffith.edu.au

Chengzheng Sun
School of Computer Engineering
Nanyang Technological University
Singapore, 639798
CZSun@ntu.edu.sg

Abstract

Multi-way dataflow constraints are very useful in the development of collaborative applications, such as collaborative CAD and CASE systems, but satisfying multi-way dataflow constraints in the presence of concurrency in collaborative systems is difficult. In this article, we discuss the issues and techniques in maintaining multi-way dataflow constraints in concurrent environments. In particular, we also proposed a novel strategy that is able to reconstruct computation flows to satisfy multi-way dataflow constraints according to concurrent user operations in collaborative systems. Our strategy ensures both constraint satisfaction and system consistency, which is independent of the execution orders of concurrent operations.

1. Introduction

A constraint specifies a relation or condition that must be maintained in a system. A dataflow constraint is an equation that has one or more methods associated with it that may be used to satisfy the equation [19]. Dataflow constraints are used in many applications, such as graphic editing systems, graphical interface toolkits, spreadsheets, simulation systems, etc. They are capable of expressing relationships over multiple data types and are conceptually simple [8], [15], [19].

Multi-way constraints have a number of advantages over one-way ones [8], [15], [19]. They provide a flexible measure to maintain relations or conditions among constrained variables, which ensures the constraints be satisfied in multi-direction.

Multi-way dataflow constraints are very useful in the development of concurrent interactive applications, such as collaborative CAD (computer aided design) and CASE (computer aided software engineering) systems. They can be adopted to confine and coordinate concurrent operations.

A task demanding people to work collaboratively is often complex and may contain many constraints. Thus, it is very practical and powerful for collaborative systems to maintain constraints automatically on behalf of users. For example, when people work collaboratively to design a project using Java Class notation, many conflicts may arise if a system only relies on individuals to maintain Java single inheritance constraint.

On the other hand, satisfying multi-way dataflow constraints in collaborative systems is very difficult. Concurrent operations may result in some constraints becoming difficult to satisfy even though they may be maintained easily in single user environments. For example, it is hard for us to satisfy the constraint defining "$X=Y+Z$", when three users concurrently change X, Y and Z respectively. In addition, interferences among constraints may be very intricate and difficult to coordinate in collaborative systems.

Much work has been done on the maintenance of multi-way dataflow constraints in single user interactive applications [8], [12], [15], [19]. However, maintenance of constraints in concurrent environments has many new features which cannot be handled by single user strategies, such as ensuring both constraint satisfaction and system consistency, handling the constraint violations generated by concurrent operations, etc.

In this paper, we discuss the issue of maintaining constraints in collaborative environments and propose a novel strategy to satisfy multi-way dataflow constraints according to concurrent user operations in collaborative systems. Our strategy ensures both constraint satisfaction and system consistency, which is independent of the execution orders of concurrent operations.

The rest of this article is organized as follows. Section 2 introduces the issue of maintaining multi-way dataflow constraints in collaborative systems. We discuss problems of constraint maintenance in collaborative environments and propose a novel strategy that is able to reconstruct computation flows to satisfy a set of predefined multi-way dataflow constraints according to concurrent user operations in collaborative systems. Comparison with related work is introduced in section 3 and the major contributions and future work of our research are summarized in section 4.

2. Maintaining constraints in concurrent environments

1-4244-0030-9/05/$20.00 ©2005 IEEE

2.1. Multi-way dataflow constraint

A dataflow constraint is an equation that has one or more Constraint Satisfaction Methods (CSM) associated with it that may be used to satisfy the equation [19]. For example, a constraint C representing the relationship "$X=Y+Z$" is a dataflow constraint. C can be satisfied only in one direction (i.e. C is a one-way constraint), if there is only one CSM associated with it, such as "$X \leftarrow Y+Z$" which means X should be calculated according to Y and Z. Here, both Y and Z are input variables and X is the output variable of C. On the other hand, a multi-way constraint in general has a method for calculating a value for each of the variables it constrains, in terms of the values of the other variables [15]. Thus, multi-way constraints can be satisfied in multi-direction. In the above example, if C is a multi-way constraint, C can be satisfied by applying the other two CSMs, "$Y \leftarrow X-Z$" and "$Z \leftarrow X-Y$", as well.

Dataflow constraints are commonly expressed in terms of constraint graphs, such as figure 1(a) that represents two constraints. C_1 defines "$W=X+V$" and C_2 constrains "$X=Y+Z$". In this and subsequent figures, a circle represents a variable and a square expresses a constraint. Initially, the constraint system is represented as an undirected bipartite graph, such as figure 1(a), where a set of undirected edges denote the relationships between variables and constraints [8], [12], [15], [19]. If a constraint satisfaction method M is selected to satisfy constraint C, all the inputs to M are represented as directed edges from the input variables to C and a directed edge from C points to M's output. In figure 1(b), Y and Z are the inputs and X is the output of constraint C_2.

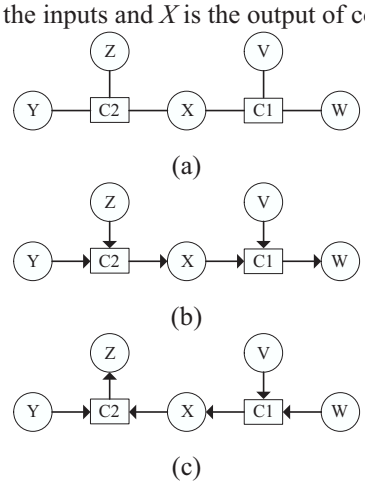

Figure 1. Graphic representations of two constraints: initially the graph is undirected as in (a). The directed graphs in (b) and (c) are solution graphs of the two constraints

A set of constraints, $CS=\{C_1, C_2, \ldots, C_n\}$, are satisfiable if for each $C_i \in CS$, $1 \le i \le n$, a method M can be selected to satisfy it, such that (1) all satisfiable constraints and their variables form a directed, acyclic graph and (2) no variable in the graph can be pointed to by more than one directed edge (i.e. a variable can be the output of at most one constraint in the graph). A direct graph that satisfies these two conditions is called a solution graph [19].

A solution graph represents a computation flow to satisfy a set of constraints. Different solution graphs can be formed to satisfy the same set of constraints. For example, to satisfy two constraints C_1: "$W=X+V$" and C_2: "$X=Y+Z$", the computation flows shown in both figure 1(b) and figure 1(c) can be adopted.

User operations may change constrained variables in interactive systems. Suppose a solution graph, SGi, represents the initial computation flow to maintain all the satisfiable constraints of a system. If a user operation modifies a constrained variable V_i, to satisfy the downstream constraints of V_i, the outputs of these constraints should be recalculated according to the new value of V_i. For any constraint C in the current solution graph, if there is a directed path from V_i to C, C is a downstream constraint of V_i. For example, in figure 1(b), both C_1 and C_2 are downstream constraints of Y. Thus, when Y is changed by a user operation, X and W that are the outputs of C_2 and C_1 should be recalculated to satisfy the constraints.

On the other hand, we cannot satisfy all the upstream constraints of V_i according to SGi. A constraint C is an upstream constraint of V_i, if there is a directed path from C to V_i in the current solution graph. For example, in figure 1(b), C_2 is an upstream constraint of variable X. When X is changed by a user operation, O, it is determined by O rather than by C_2. Therefore, the initial computation flow, where X is the output of C_2, cannot be applied to satisfy the constraint. To satisfy C_2, we can change the computation flow of C_2 or abort/block the user operation. Satisfying constraints only by aborting/blocking user operations is undesirable, because it destroys users' work and degrades or even breaks the interactions between users and applications.

A significant merit of multi-way constraints is that they can be satisfied in multi-direction. Therefore, when user operations change constrained variables, it is desirable to maintain constraints by reconstructing their computation flows, which can retain operations' effects.

For a set of satisfiable constraints, $CS=\{C_1, C_2, \ldots, C_n\}$, the reconstruction of the computation flow for any $C_i \in CS$, $1 \le i \le n$, may cause the computation flow of another constraint to be reconstructed, propagating to further constraints. For example, in figure 1(b), a user operation that changes W triggers the reconstruction of the computation flow for C_1. If X is selected as the new output of C_1, the computation flows of C_1 and C_2 conflict with each other, because X is the output of both constraints. To solve this problem, we may change the

computation flow of C_2 as well. After the reconstructions of the computation flows of both constraints, we may obtain a new solution graph as shown in figure 1(c).

In collaborative systems where user operations may be generated concurrently, reconstructing constraints' computation flows is difficult. For example, it is hard for us to determine the computation flow to satisfy a horizontal-line constraint, which requires that the y values of the both endpoints of a horizontal-line should be equal, when two users concurrently change both endpoints of a horizontal-line to different vertical positions from different sites.

In this paper, we focus on the issue of reconstructing computation flows to maintain a set of predefined satisfiable constraints when concurrent user operations modify the constrained variables in collaborative systems. Here, a set of predefined satisfiable constraints means (1) which constraints should be maintained in a system has been determined and these constraints have been satisfied on the initial document state, and (2) no constraint is added to or deleted from the system after the execution of the first user operation. Therefore, user operations, which modify constrained variables, cannot interfere with the additions and deletions of constraints.

Moreover, our research is based on acyclic constraint graph, which means that there is not any cycle in the undirected bipartite graph that represents the relations of the predefined constraints and their variables. Maintaining cyclic constraints in collaborative systems can be very difficult, which is beyond the scope of this paper and will be discussed in our subsequent publications.

2.2. Problems of maintaining multi-way dataflow constraints in collaborative systems

Collaborative systems are groupware applications to support people working together in groups, such as electronic conferencing/meeting, collaborative CAD and CASE [16], [17]. Multi-way dataflow constraints are very useful in collaborative systems, which can confine and coordinate concurrent operations. For example, in a collaborative spreadsheet system, dataflow constraints may represent the relationship among different cells. Thus, users can concurrently input data to different cells from different sites and the underlying system maintains the relationship automatically.

To meet the requirement of high responsiveness in the Internet environment, replicated architecture is widely adopted in collaborative systems. Shared documents are replicated at the local storage of each collaborating site, so that operations can be performed at local sites immediately and then propagated to remote sites [1], [16]. However, maintaining consistency among replicas is more complex than sharing a single copy of centralized data, especially in collaborative systems with constraints, which is illustrated in the following scenario:

Scenario 1. Constraint C_1 defines "*middle-point=(left-point+right-point)/2*". Two users generate operations changing the positions of *middle-point* and *left-point* at the same time from different sites.

Figure 2 represents scenario 1. The initial computation flow of C_1 is shown in rectangular boxes 1.0 and 2.0. Two concurrent operations are generated in the scenario: O_1= *Move(left-point,* P_a*)* by user 1, and O_2= *Move(middle-point,* P_b*)* by user 2. At the site of user 1, *left-endpoint* is first moved to position P_a, resulting in the recalculation of *middle-point.* The solution graph of C_1 will not be modified, because C_1 is a downstream constraint of *left-endpoint*, as shown in rectangular box 1.1. When O_2 arrives and is executed at user 1's site, it will invoke the reconstruction of C_1's computation flow. Two possible solution graphs are shown in rectangular boxes 1.2 and 1.3. At the site of user 2, after the execution of O_2, two possible computation flows to satisfy C_1 are shown in rectangular boxes 2.1 and 2.2. Applying O_1 to these two solution graphs, we can obtain two possible computation flows of C_1, as shown in rectangular boxes 2.3 and 2.4.

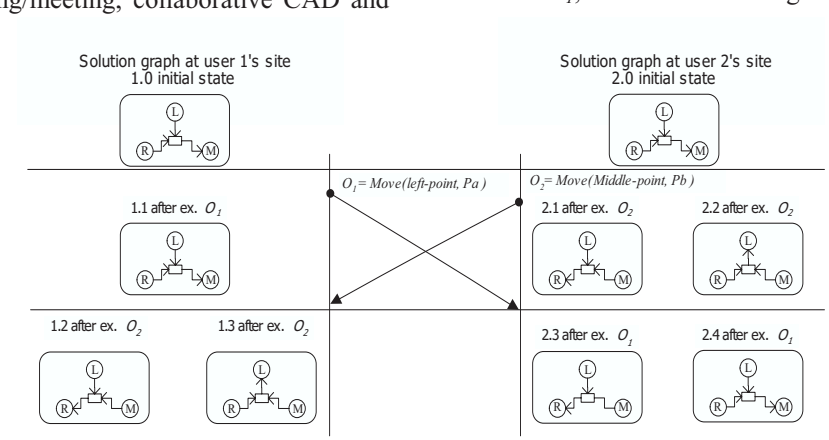

Figure 2. Maintenance of a constraint generates divergence in collaborative environments

In scenario 1, after executing the two operations in different orders, we can obtain three possible computation flows to satisfy C_1. Therefore, it is very likely to generate divergence. In this example, the computation flow shown in rectangular box 1.2 (identical to the one shown in rectangular box 2.3) is preferable, because it retains the effects of both user operations.

Many problems in collaborative systems are derived from concurrent operations. Therefore, serialization undo/redo strategy is often adopted in collaborative systems. This strategy maintains consistency by ensuring that operations are executed in the same order at each site according to their total ordering relation [16], [17]. Nevertheless, applying undo/redo strategy to satisfy constraints has many demerits:

First of all, it is very difficult to implement undo/redo strategy in collaborative systems with multi-way dataflow constraints. For example, in scenario 1, suppose O_1 total ordering precedes O_2. When O_1 arrives at user 2's site, O_2 should be undone. However, undoing O_2 is not so easy a job as to restore the value of *middle-point*, because the computation flow of C_1 is reconstructed after the execution of O_2. If a system contains many constraints, it is complicated and difficult to restore a solution graph to its previous states.

Moreover, even if we can ensure that concurrent operations are executed in the same order at different sites, the problems of scenario 1 still cannot be solved. For instance, even though O_1 is executed before O_2 at each site, we still cannot ensure that each site will obtain the preferred solution graph shown in rectangular box 1.2 of figure 2. They may also get the result shown in rectangular box 1.3.

Finally, this strategy degrades the performances of collaborative systems. If an operation with a smaller timestamp is delayed, we may have to undo and redo many operations to execute the operation. Interactive applications need efficient performance to meet the demands of real-time direct manipulation. Therefore, it is undesirable to adopt this strategy to maintain multi-way dataflow constraints in collaborative systems.

Our challenge is to devise a strategy that is able to reconstruct computation flows to maintain a set of predefined satisfiable constraints according to concurrent user operations in collaborative systems adopting replicated architecture, and the strategy should have the following features: (1) maintaining both satisfiable constraints and system consistency, (2) independent of the execution orders of concurrent operations, and (3) taking the effects of concurrent operations into account.

2.3. Timestamp constrained variables to compare priorities

Priority strategies are widely adopted in collaborative systems [11], [18], which maintain consistency according to the priorities instead of the execution orders of concurrent operations. They can be applied to handle conflicts and maintain constraints in collaborative environments.

When an operation O, which intends to modify variable V, is ready for execution at site A, if the current value of V is the execution effect of O_i, (we say O_i determines V), the execution of O will mask the effect of O_i. If O and O_i are concurrent operations, they conflict with each other and it is likely that O is executed before O_i at site B. Thus, the effect of O will be masked after the execution of O_i at site B and divergence occurs.

To solve this problem without undoing/redoing operations, a priority strategy is adopted. If there is a conflict in retaining all operations' effects in collaborative applications, the strategy masks operations' effects according to their priorities. For instance, in the above example, if the priority of O is higher than the priority of O_i, O_i will be masked at each site.

Applying this priority strategy, the correct masking result is independent of the execution orders of the concurrent operations which conflict with each other: If concurrent operations are executed in the ascending order of their priorities, the correct masking result can be achieved by their natural masking effects. On the other hand, if an operation with a higher priority is executed before an operation with a lower priority, the correct masking result can be achieved simply by voiding the lower priority operation. For example, if O, which has a higher priority, is executed before O_i at site B, O_i will not have any effect at the site.

In this paper, the timestamp of an operation denotes its priority, the bigger the timestamp the higher the priority. Let n be the number of cooperating sites in a system. Each site maintains a State Vector (SV) with n components. Initially, $SV[i]=0$, for all $1 \le i \le n$. After executing an operation generated at site i, $SV[i]=SV[i]+1$. An operation is executed at the local site immediately after its generation and then multicast to remote sites with a timestamp of the current value of the local SV [16]. In this paper, the value of timestamp SV_O is defined as $SV_O[1]+ SV_O[2]+\ldots+ SV_O[n]$. For any two operations O_a and O_b, the timestamp of O_a is bigger than the timestamp of O_b, if and only if O_b total ordering precedes O_a [16].

To achieve the correct masking result efficiently, we introduce the concept of the timestamp of a variable, which is the association of a variable with operation-timestamp:

If user operation O_i changes variable V (V is determined by O_i), the timestamp of V is the timestamp of O_i.

For example, if V is determined by O_i when O, intending to change V, is ready for execution at a site, the

timestamp of V must be the timestamp of O_i. Thus, whether O should be masked can be decided simply by comparing the timestamps of O and V.

It is also possible that variable V is determined by a constraint rather than by an operation. For instance, V is the output of constraint C defining *"V=X+Y"*. To satisfy C, either X or Y should be selected as the new output of C when V is changed by O and becomes an input of C (i.e. X or Y should be changed to satisfy C when a user changes V). If X is selected as the new output and it is previously determined by operation O_j, the effect of O_j will be masked, because X is determined by constraint C rather than by O_j after the execution of O. If O_j's priority is higher than O's priority, the outcome cannot satisfy the priority strategy which masks the operations with lower priorities if there is a conflict of retaining all operations' effects in collaborative systems with constraints. Thus, system consistency cannot be ensured if these operations are executed in different orders at different sites. To achieve the correct masking result, before executing O, the timestamps of O, X and Y should be compared (Here, we suppose that both X and Y are determined by user operations). O can be executed only if its timestamp is bigger than the timestamp of an input of C. If O can be executed, the one with the smallest timestamp amongst all the inputs of C should be selected as the new output of C, and the operation previously determines it will be masked automatically when the value of the new output is recalculated according to the inputs of C.

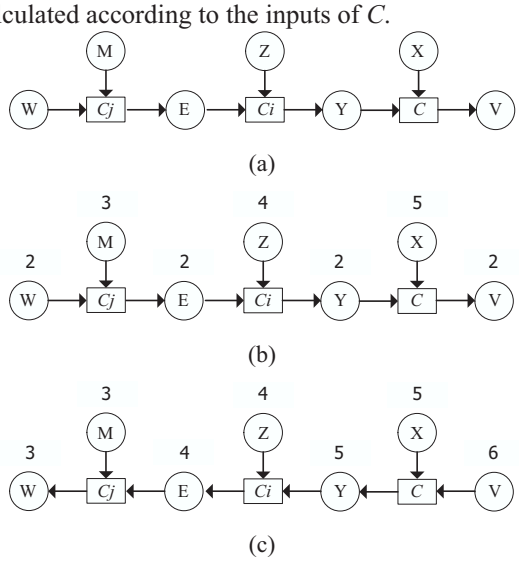

Figure 3. Graphic representations of three constraints: one of the inputs of *C* is determined by C_i in (a), timestamp value of each variable before and after the execution of *O* is represented above the variable's name in (b) and (c).

Some inputs of constraint C may be determined by other constraints. For example, in figure 3(a), Y is an input of C and the output of C_i (i.e. Y is determined by C_i). Thus, before executing O, which intends to modify V in figure 3(a), the timestamp of O and the timestamps of the inputs of both constraints C and C_i should be compared. If some inputs of C_i are determined by other constraints, such as C_j determining E, the timestamps of the inputs of these constraints should also be compared. This process continues until the timestamps of all the inputs of all the upstream constraints of V are compared. It may be very time consuming.

Comparing the timestamps of the inputs of a constraint, C, is to find the input, V_i, whose timestamp is the minimum of the timestamps of all the inputs of the constraint. V_i will be selected as the output of C when the current output of the constraint is changed by a user operation. Therefore, we can define the timestamp of the output of constraint C according to the timestamp of V_i:

If variable V is the output of method M of constraint C (V is determined by C), V's timestamp is the minimum of the timestamps of M's inputs.

Let V be the output of constraint C, whether operation O, intending to change V, should be masked can be easily determined by comparing the timestamps of O and V. Thus, it is unnecessary to compare the timestamps of all the inputs of all the upstream constraints of V. For example, suppose in figure 3(b), W, M, Z and X, are determined by user operations O_1, O_2, O_3 and O_4 respectively. Their timestamp values are 2, 3, 4, and 5. E is determined by constraint C_j. Therefore the timestamp of E is set as the timestamp of W whose timestamp is the minimum of the timestamps of all the inputs of C_j. For the same reason, the timestamp values of both Y and V that are the outputs of C_i and C, are set as 2. Operation O, which intends to modify V in the current solution graph, can be executed only if its timestamp is bigger than the timestamp of V. Suppose the timestamp value of O is 6 in this example. O will be executed and V's timestamp will be set as the timestamp of O, because V is determined by O rather than by C after the execution of O. To satisfy C, Y that has the smallest timestamp amongst all the inputs of C, will be selected as the new output of C. Both the timestamp and value of Y should be reset according to the current inputs of C. Y is the output of constraint C_i previously, when it becomes the output of C, the computation flow of C_i should be reconstructed. Otherwise, the computation flows of C and C_i conflict with each other. Accordingly, E that has the smallest timestamp amongst all the inputs of C_i should be changed to the new output of C_i. This process continues until W, which is not determined by any constraint previously, is selected as the new output of C_j. After O is executed and the computation flows of the three constraints are reconstructed, the final solution graph and the timestamp value of each constrained variable are shown in figure 3(c). In the final solution graph, O_2, O_3, and O_4 still

determine *M, Z,* and *X*, but O_1, which determines *W* previously and has the smallest timestamp amongst all the operations, is masked. The outcome achieves the correct masking effect.

2.4. A computation flow reconstruction strategy

Based on the timestamp information of constrained variables, our Computation Flow Reconstruction strategy (CFR) can be represented as follows:

If a user operation *O* intends to change a constrained variable *V* in a solution graph, *SGo*, the timestamps of *V* and *O* will be compared. If the timestamp of *V* is bigger than the timestamp of *O*, *O* cannot have any effect on the current document state. Otherwise, *O* will be executed and the timestamp of *V* will be set as *O*'s timestamp. To maintain the downstream and upstream constraints of *V* in *SGo*, procedures Down_flow( ) and Up_flow( ) will be invoked in sequence.

```
Procedure Down_flow(V)
For each constraint C while V is an
input of C, do
(1) Let Vo be the output of C, reset
   the timestamp of Vo
(2) Recalculate the value of Vo
(3) Call Procedure Down_flow(Vo)
```

After *O* changes *V*, for each constraint *C* while *V* is an input of it, *C*'s output, V_o, should be recalculated and the timestamp of V_o should be reset, because both the value and the timestamp of *V* which is an input of *C* are changed. The effects of modifying the timestamp and value of V_o should be propagated to V_o's downstream constraints.

```
Procedure Up_flow(V)
If V is the output of constraint C in
SGo, then
(1) Set variable Vm, an input of C,
   whose timestamp is the minimum of
   the timestamps of all the inputs of
   C, as the output of C
(2) Set V as an input of C
(3) Reset the timestamp of Vm
(4) Recalculate the value of Vm
(5) Call Procedure Down_flow(Vm)
(6) Call Procedure Up_flow(Vm)
```

In the above algorithm, if *V* is the output of constraint *C* in *SGo*, after the execution of *O*, which modifies *V*, *V* is changed to an input of *C*, and V_m, an input of *C* in *SGo*, whose timestamp is the minimum of the timestamps of all the inputs of *C,* is set as the output of *C*. Accordingly, the timestamp of V_m should be reset and the value of V_m should be recalculated according to the current inputs of *C*. The effects of the modification of V_m should be propagated to all the downstream constraints of V_m and the constraint that previously determines V_m in *SGo*. This process continues until an input is selected as the new output of an upstream constraint of *V* in *SGo*, and this input is not determined by any constraint in *SGo*.

The proposed CFR strategy reconstructs the computation flows for a set of predefined constraints according to the timestamps of constrained variables and user operations, which is independent of the execution orders of concurrent operations. For a group of concurrent operations, $OG=\{O_1, O_2,\ldots,O_n\}$, applying to *k* multi-way dataflow constraints, $C_1, C_2,\ldots, C_k$, on document state *DSo,* if each input of any C_i, $1\leq i\leq k$, has a unique timestamp on *DSo*, applying CFR strategy, we will obtain a unique solution graph SG_1 after executing all the concurrent operations in *OG* in any order. For each constrained variable *V* in SG_1, if *V* is determined by operation $O_i\in OG$, $1\leq i\leq n$, O_i's timestamp is the maximum of the timestamps of all the operations in *OG* which target *V*. Otherwise, if *V* is determined by a constraint, the timestamp of *V* in SG_1 is bigger than the timestamp of any operation in *OG*, which intends to change *V*. Thus, which operations have effects on the final document state is independent of the execution orders of these concurrent operations. The operations, which have effects on the final document state, determine the inputs of the constraints. Thus, the final value of each constrained variable is also independent of the execution orders of concurrent operations.

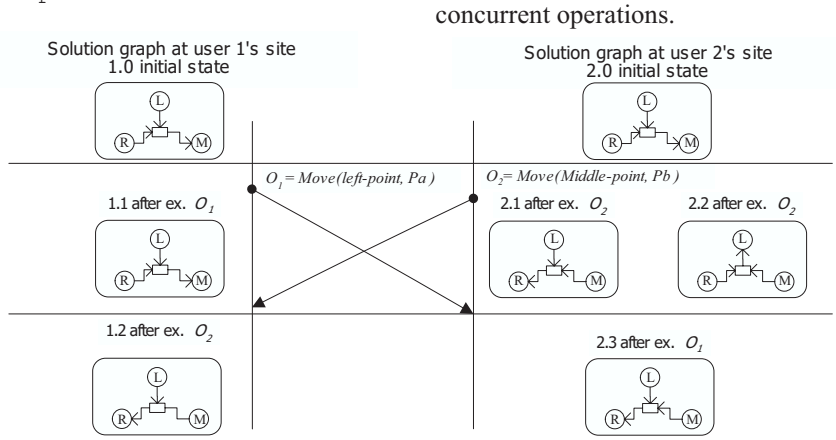

Figure 4. Maintaining both constraint and consistency

The effects of the proposed strategy can be illustrated by following examples:

In scenario 1, suppose the timestamps of O_1 and O_2 are T_1 and T_2 ($T_1>T_2$). The initial timestamp of *left-endpoint*, *right-endpoint* and *middle-point* are T_l, T_r and T_m respectively, which are smaller than T_1 and T_2.

At user 1's site, the timestamp of *left-endpoint* is T_1 after the execution of O_1. When O_2 is executed at the site of user 1, *middle-point* becomes an input of C_1. Thus, the computation flow of C_1 should be reconstructed. After the execution of O_2, the timestamp of *middle-point* is set as T_2. Therefore, *right-point* will be selected as the new output, because its timestamp is smaller than the timestamps of both *left-endpoint* and *middle-point*. The final computation flow of C_1 at user 1's site is shown in rectangular box 1.2 of figure 4.

At user 2's site, if $T_r<T_l$, after the execution of O_2, we obtain a solution graph as shown in rectangular box 2.1 of figure 4. Otherwise, if $T_r>T_l$, the solution graphic is shown in rectangular box 2.2. Under both conditions, the timestamp of *middle-point* is set as T_2 after the execution of O_2.

When O_1 is executed at user 2's site, if the current solution graph is in the state shown in rectangular box 2.1, the execution of O_1 will not cause the reconstruction of C_1's computation flow. Thus, after executing O_1 at user 2's site, we obtain a solution graph shown in rectangular box 2.3 of figure 4. On the other hand, if O_1 is applied to the solution graph shown in rectangular box 2.2, the execution of O_1 will change C_1's computation flow. Under this condition, only *right-endpoint* can be selected as the output of C_1, because the timestamps of both *left-endpoint* and *middle-point* are bigger than the timestamp of *right-endpoint* after the executions of both operations. Thus, the final solution graph of C_1 at the site of user 2 is shown in rectangular box 2.3, which is identical with the final solution graph at the site of user 1. Therefore, both consistency and constraint are maintained

Scenario 2, the initial solution graph of three constraints and the timestamp value of each constrained variable are shown in figure 5(a). Two users concurrently execute operations O_1 and O_2, which change V and W respectively. The timestamp values of O_1 and O_2 are 6 and 7.

At user 1's site, the two operations are executed in the order O_1, O_2:

(1) When O_1 is executed, V becomes an input of C, because it is determined by O_1 rather than by C. Thus the timestamp of V is set as the timestamp of O_1. Then, Y is selected as the new output of C, because its timestamp is the minimum of the timestamps of all the inputs of C. After Y is set as the output of C, the timestamp of Y is reset and the value of Y is recalculated, which triggers the reconstruction of the computation flow for C_i, and results in that E becomes the new output of C_i. The change of the computation flow of C_i triggers the modification of the computation flow for C_j, and W is selected as the new output. Thus, after O_1 is executed, the solution graph is shown in figure 5(b),

(2) When O_2 is executed, the computation flow of C_j is reconstructed. M becomes the output and W is changed to an input of C_j, as shown in figure 5(c).

At the site of user 2, the two operations are executed in the order O_2, O_1:

(1) When O_2 changes W in the solution graph shown in figure 5(a), the outputs of all the downstream constraints of W should recalculate their values and reset their timestamps. Because W is not the output of any constraint in the initial solution graph, no constraint's computation flow will be modified, as shown in figure 5(d),

(2) When O_1 is applied to the solution graph shown in figure 5(d), Y, E, and M, which are the inputs with the smallest timestamps of C, C_i and C_j, will be selected as the new outputs of the three constraints. After the execution of O_1, the new solution graph is shown in figure 5(c).

After executing the two operations in different orders at different sites, we obtain the identical solution graphs to satisfy the three constraints.

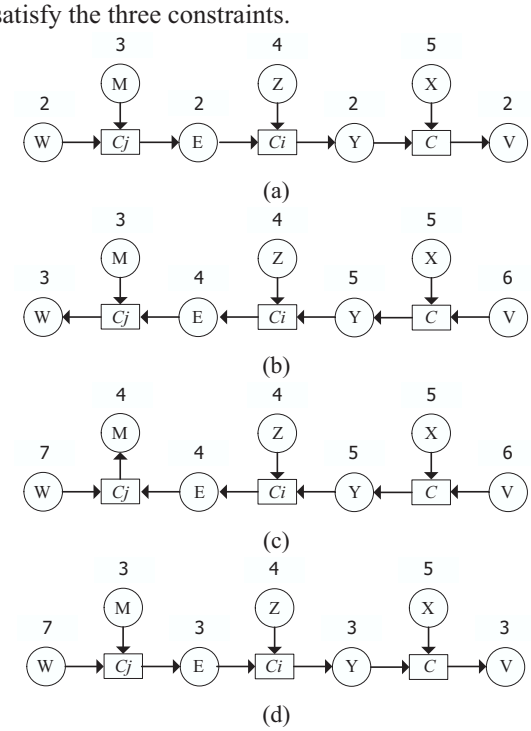

Figure 5. The solution graphs of three constraints, the timestamp value of each variable is represented above the variable's name.

2.5. Freshness of constrained variables

If many inputs of a constraint, C, have the same timestamp, applying the proposed CFR strategy to reconstruct the computation flow for C may generate divergence. For example, in scenario 1, suppose on the initial document state the timestamp of each constrained variable is T, which is smaller than the timestamps of both O_1 and O_2. After executing O_2 at the site of user 2, we may obtain two possible solution graphs, as shown in rectangular boxes 2.1 and 2.2 of figure 2. Even if only one operation is executed, we get two possible solution graphs. Therefore, system consistency cannot be ensured.

The correctness of CFR strategy is based on the precondition that each input variable of a constraint has a unique timestamp. Every user operation has a unique timestamp. If an input variable of C is modified by a user operation O, its timestamp will be set as the timestamp of O. Therefore, its timestamp must be different from the timestamps of the other inputs of C. However, on the initial document state where no user operation has been executed, the timestamp of each constrained variable is null. Thus, the precondition, which requires a unique timestamp of each input of a constraint, cannot be satisfied. Under this condition, extra information should be provided to ensure the correctness of the proposed CFR strategy. The following approach is straightforward:

Each constraint associates each of its constrained variables, V, with a unique variable number, denoting as $C.Vn(V)$, which means the variable number of V defined by constraint C. When the computation flow of constraint C is to be reconstructed, if the timestamps of many inputs of C are null, the one with the smallest variable number will be selected as the output of C.

For a single constraint, variable numbering approach works. However, if a variable relates to more than one constraint, it may have many variable numbers, each number corresponding to a constraint, such as in figure 6, both C_1 and C_2 may define the variable number of V_b, so that V_b does not have a unique variable number. It is difficult to reconstruct the computation flows to maintain the constraints and consistency under this condition.

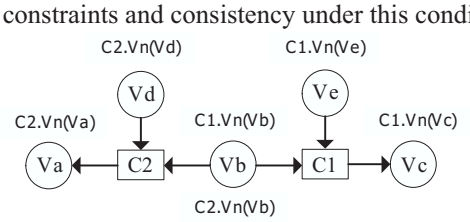

Figure 6. Graphic representation of two constraints, the variable number of each variable is represented above/below the variable's name

The above problem can be avoided if a variable can only be numbered by at most one constraint. However, comparing the variable numbers defined by different constraints is still a challenge. For instance, suppose in figure 6, V_b is only numbered by C_2 while V_e is numbered by C_1. When a user operation changes V_c, we should determine the output of C_1 by comparing the variable numbers of V_b and V_e, but it is possible that the values of $C_1.Vn(V_e)$ and $C_2.Vn(V_b)$ are equal, then, how can we compare the variable numbers defined by different constraints?

To handle the above problems, we can associate each predefined constraint with a unique constraint number. A constraint number denotes the importance of its associated constraint, the bigger the constraint number, the more important the constraint. Accordingly, when a user operation changes V_c in figure 6, rather than comparing the values of $C_1.Vn(V_e)$ and $C_2.Vn(V_b)$, we compare the constraint numbers of C_1 and C_2 that define the variable numbers of V_e and V_b respectively. If the constraint number of C_1 is bigger than the constraint number of C_2, V_b will be selected as the output of C_1. Otherwise, V_e will be the new output of C_1.

Based on the above discussion, we define the *freshness* of a constrained variable, which is used to ensure the correctness of the proposed CFR strategy:

The freshness of a constrained variable is a 3-ary tuple, (T, Cn, Vn). T is the timestamp of the variable. Cn and Vn are constraint number and variable number of the variable.

If variable V is the output of method M of constraint C (V is determined by C), V's freshness is the minimum of the freshness of M's inputs.

For two variables V_1 and V_2, the *freshness* of V_1, (T_1, Cn_1, Vn_1), is bigger than the *freshness* of V_2, (T_2, Cn_2, Vn_2), if (1) neither T_1 nor T_2 is null, and $T_1>T_2$, or (2) T_1 is not null but T_2 is null, or (3) both T_1 and T_2 are null, and $Cn_1>Cn_2$, or (4) both T_1 and T_2 are null, $Cn_1=Cn_2$, and $Vn_1>Vn_2$.

Accordingly, we extend the CFR strategy to handle the problem that more than one inputs of a constraint have null timestamps: When an operation O changes a constrained variable V, the computation flows of V's upstream constraints should be reconstructed according to the *freshness* rather than the timestamps of the constrained variables. In addition, the *freshness* of the outputs of all the downstream constraints of V should be reset according to the current *freshness* of their inputs.

Suppose on the initial document state, the solution graph to satisfy a set of predefined satisfiable constraints is *SGo*, to maintain these constraints using CFR strategy, we should initialize the *freshness* of each constrained variable, so that (1) the *freshness* of a variable cannot be defined by more than one constraint, (2) the *freshness* of any input of any constraint C should be different from the *freshness* of the other inputs of C, and (3) for the output of a constraint, its *freshness* should be the minimum of the *freshness* of all the inputs of all its upstream constraints.

The above results can be achieved by applying the following *freshness* initialization strategy:

(1) In constraint number descending order, find a constraint C in SGo, while no input of C is determined by any other constraint,

(2) For each V_i, which is an input of C, if the *freshness* of V_i has not been initialized, initialize the *freshness* of V_i as ($null, Cn, C.Vn(V_i)$), where Cn is the constraint number of C and $C.Vn(V_i)$ is the variable number of V_i defined by constraint C,

(3) After the *freshness* of all the inputs of C has been initialized, set the *freshness* of the output of C as the minimum of the *freshness* of C's inputs,

(4) For any V, a constrained variable of C, if V is not associated with any other constraint in SGo, delete V from SGo,

(5) Delete C from SGo,

(6) Repeat the above process, until all the constraints and their constrained variables are eliminated from SGo.

According to the above approach, the constrained variables of constraint C_i can be initialized only if (1) no input of C_i is determined by any other constraint in the initial solution graph, or (2) every constraint which determines an input of C_i in the initial solution graph has been deleted from SGo (i.e. their constrained variables have been initialized). Because our discussion is based on acyclic constraint graph, at least one constraint can satisfy the above condition (1) on the initial document state and the above process terminates only when all the constraints have been eliminated from an acyclic solution graph.

Initializing the *freshness* of constrained variables by the above mechanism, the proposed CFR strategy is able to reconstruct computation flows to satisfy a set of predefined multi-way dataflow constraints according to user operations. Both constraint satisfaction and system consistency are maintained while concurrent operations are allowed to be executed in any order.

3. Related work

There is a large body of research efforts contributing to dataflow constraint maintenance in single user interactive applications [3], [4], [8], [12], [15], [19].

Many approaches are based on constraint hierarchies and *walkabout* strength strategy [4], [8], [15]. Constraint hierarchies denote the importance of each constraint in different levels, such as *required, strong, medium* and *weak,* etc. Each variable is associated with *walkabout* strength information. When a constraint is added to a system, this information is used to determine whether or not to enforce the constraint and what other constraint should be unenforced to avoid a conflict [8].

Walkabout strength strategy determines computation flows incrementally by comparing constraints' importance. Therefore, to change the value of a constrained variable V to v, *walkabout* strength strategy imposes a new constraint C to V, which confines the value of V equals to v. C must be defined important enough to change the value of V which is previously confined by other constraints. However, in collaborative environments, updating variables by adding new constraints has many demerits. First of all, adding new constraints may cause other constraints to become unsatisfiable. When users change constrained variables by imposing new constraints, it is common that each user sets his/her constraints as the most important ones to avoid them to be masked by the constraints defined by other users. This may cause some predefined constraints become unsatisfiable, which is contradictory to our intention that applying constraints to confine and coordinate user operations in collaborative systems. Moreover, when users concurrently add constraints to a system, *walkabout* strength strategy cannot ensure system consistency.

Doppler [2] and CAB [10] are related to constraint control in collaborative applications. Doppler supports distributed, concurrent, one-way constraints in user interface applications. Doppler algorithm can be applied to systems where constraint solution graphs are distributed, but consistency maintenance in collaborative systems adopting replicated architecture is beyond its concern. Doppler only maintains one-way constraints. Therefore, the solution graph to satisfy constraints is predefined and cannot be changed by user operations that modify constrained variables.

CAB presents an active rule based approach to modeling user-defined semantic relationships in collaborative applications and explores a demonstrational approach for end-user customization of collaboration tools to support the definition of those relationships. Constraints in CAB include those for coordination between distributed users such as awareness, access, and concurrency control, which are beyond the scopes of graphic objects [10]. However, just as its author stated, many complications of maintaining constraints in collaborative environments, such as how to handle constraint violations and coordinate interferences among constraints, are not investigated in CAB.

In comparison with the above research efforts, we have focused on the issues and techniques in maintaining multi-way dataflow constraints in collaborative systems and proposed a novel strategy that is able to reconstruct computation flows to satisfy a set of predefined multi-way dataflow constraints according to concurrent user operations in collaborative environments. In our approach, user operations will not change the importance of any constraint and cannot cause any predefined satisfiable constraint to become unsatisfiable. Our strategy ensures both constraint satisfaction and system

consistency, which is independent of the execution orders of concurrent operations.

4. Conclusion and future work

Multi-way dataflow constraints are very useful in collaborative systems, which can confine and coordinate concurrent operations. However, satisfying multi-way dataflow constraints in the presence of concurrency in collaborative systems is a challenging task. The difficulties are caused by concurrent operations that modify constrained variables and cause the reconstructions of the computation flows for constraints. Being able to solve this problem is crucial in the development of collaborative applications, such as collaborative CAD and CASE systems.

In this paper, we propose a computation flow reconstruction strategy to solve this problem. This strategy ensures both constraint satisfaction and system consistency. Our solution does not require operations to be undone/redone to achieve convergence, as undoing and redoing operations degrades system performance and increases complexity.

We are currently investigating any-undo strategy in collaborative systems with multi-way dataflow constraints, which allows user operations be undone/redone in any order at any time. Undo is a very useful and common feature in interactive applications, but just as we discussed previously, undoing an operation may be very difficult in collaborative systems with constraints.

There are some limitations in applying our strategy to collaborative systems. For example, if constraints can be added to or deleted from a system when users are manipulating the constrained variables, divergence may occur. Moreover, our strategy lacks the ability to handle cyclic and inequality constraints. How to solve these problems is currently being investigated and will be reported in our subsequent publications.

5. References

[1] Begole James et al., "Resource Sharing for Replicated Synchronous Groupware". *IEEE/ACM Transactions on Networking*. Vol. 9, No.6, Dec. 2001, pp.833-843.

[2] Bharat, K. and Hudson, S. E, "Supporting Distributed, Concurrent, One-way Constraints in User Interface Applications", In *Proceedings of the ACM Symposium on User Interface Software and Technology*, *ACM*, New York, 1995, pp.121-132.

[3] Borning, A. and Duisberg, R., "Constraint-based Tools for Building User Interfaces", *ACM Transactions on Graphics*, Vol.5, No.4, Oct.1986, pp.345-374.

[4] Borning, A. et al., "Constraint Hierarchies", *Lisp and Symbolic Computation*, Vol.5, No.3, Sep.1992, pp.223-270.

[5] Dourish, P., "Developing a Reflective Model of Collaborative Systems", *ACM Transactions on Computer-Human Interaction*, 2(1), 1995.

[6] Dourish, P., "Consistency Guarantees: Exploiting Application Semantics for Consistency Management in a Collaborative Tookit", In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, *ACM*, New York, 1996, pp.268-277.

[7] Edwards, W.K., "Flexible Conflict Detection and Management in Collaborative Applications", In *Proceedings of the ACM Symposium on User Interface Software and Technology*, *ACM*, New York, 1997, pp.139-148.

[8] Freeman-Benson. B, et al., "An Incremental Constraint Solver", *Communications of the ACM*, 33(1), Jan. 1990, pp.54-63.

[9] Ignat, C.-L, Norrie, M.C., "Grouping in Collaborative Graphical Editors", *ACM Conference on Computer-Supported Cooperative Work*, Chicago, USA, Nov.6-10, 2004, pp.447-456.

[10] Li, D. and Patrao, J., "Demonstrational Customization of a Shared Whiteboard to Support User-defined Semantic Relationships amongst Objects", *ACM GROUP'01*, Boulder, Colorado, USA, Sep. 30-Oct. 3, 2001, pp.97-106.

[11] Lin, K., Chen, D. Sun C. and Dromey, R.G., "Maintaining Constraints in Collaborative Graphic Systems: the CoGSE Approach", *9th European Conference on CSCW*, Paris, France, Sep. 2005.

[12] McCartney TP., "User Interface Applications of a Multi-way Constraint Solver", *Washington University Department of Computer Science technical report WUCS-95-22*, 1995.

[13] Monfroy, E. and Castro, C., "Basic Components for Constraint Solver Cooperations", *Proceedings of SAC*, 2003.

[14] Myers, B. A., "Graphical Techniques in a Spreadsheet for Specifying User Interfaces", In *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems*, *User Interface Management Systems*, 1991, pp.243-249.

[15] Sannella, M. et al., "Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm", *Software-Practice and Experience*, Vol. 23(5), 1993, pp.529–566.

[16] Sun, C., et al., "Achieving Convergence, Causality-preservation, and Intention-preservation in Real-time Cooperative Editing Systems", *ACM Transactions on Computer-human Interaction*, 5(1), 1998, pp.63-108.

[17] Sun, C. and Chen, D., "Consistency Maintenance in Real-time Collaborative Graphics Editing Systems", *ACM Transactions on Computer-Human Interaction*, Vol. 9, No.1, 2002, pp.1-41.

[18] Sun, D. et al., "Operational Transformation for Collaborative Word Processing", *ACM Conference on CSCW*, Chicago, USA, Nov. 6-10, 2004.

[19] Zanden B., "An Incremental Algorithm for Satisfying Hierarchies of Multi-way Dataflow Constraints", *ACM Transaction on Programming Languages and Systems*, Vol.18, No.1, Jan.1996, pp.30-72.