H. Gellersen et al. (eds.), ECSCW 2005: Proceedings of the Ninth European Conference on Computer-Supported Cooperative Work, 18-22 September 2005, Paris, France, 185–204.

Maintaining Constraints in Collaborative Graphic Systems: The CoGSE Approach

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

Abstract. A constraint specifies a relation or condition that must be maintained in a system. It is common for a single user graphic system to specify some constraints and provide methods to satisfy these constraints automatically. Constraints are even more useful in collaborative systems, which can confine and coordinate concurrent operations, but satisfying constraints in the presence of concurrency in collaborative systems is difficult. In this article, we discuss the issues and techniques in maintaining constraints in collaborative systems. In particular, we also proposed a novel strategy that is able to maintain both constraints and system consistency in the face of concurrent operations. The strategy is independent of the execution orders of concurrent operations and able to retain the effects of all operations in resolving constraint violation. The proposed strategy has been implemented in a Collaborative Genetic Software Engineering system, called CoGSE, for maintaining the tree structure constraint. Specific issues related to CoGSE are also discussed in detail.

Introduction

Existing graphic systems fall into two groups, General Graphic system (GG), such as XFig, Illustrator, etc., and Special Application system (SA), such as AutoCAD, Rational Rose, etc. The former provides users with vast graphic manipulation functions and its application is broad. The latter offers sophisticated functions for specific applications, which is specially designed to maintain constraints and relations among objects in specific graphic systems. SA is advanced in satisfying constraints automatically. For instance, Rational Rose always ensures that no

cycle appears in a graph representing Class hierarchy, even though it cannot be used as a general graphic tool.

Maintaining constraints automatically is even more advantageous in collaborative systems than in single user environments, which can confine and coordinate concurrent operations. It is extremely effective to deal with complicated tasks in collaboration scenarios. 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. A task, which demands people to work collaboratively, is often complex and may contain many requirements and constraints. Thus, it is very practical and powerful for collaborative systems to maintain constraints automatically on behalf of users.

On the other hand, satisfying constraints in the presence of concurrency in collaborative systems is 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, a horizontal-line constraint, which requires that the y values of the both endpoints of any horizontal-line should be equal, is difficult to maintain when two users concurrently change both endpoints of a horizontal-line to different vertical positions. In addition, interferences among constraints may be very intricate and difficult to coordinate in collaborative systems.

Collaborative graphic systems satisfying constraints automatically are rare, even though much work has been done on collaborative graphic systems (Sun and Chen, 2002, Ignat and Norrie, 2004). In these systems, graphic objects are independent of each other and no constraint is maintained. These collaborative graphic systems provide flexible functions for users to represent arbitrary graphic notations, but they lack constraint maintenance functions. Hence, they are not applicable to complex collaborative graphic applications, such as collaborative CAD and CASE.

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 (Sun *et al*, 1998, Begole *et al,* 2001). However, maintaining consistency among replicas is more complex than sharing a single copy of centralized data, especially in collaborative systems with constraints.

The objective of this paper is to analyze constraint maintenance in collaborative environments and devise strategies to achieve both constraint satisfaction and system consistency in collaborative graphic systems adopting replicated architecture.

The rest of this article is organized as follows. The next section introduces constraint maintenance. We discuss problems of constraint maintenance in

collaborative systems and propose a novel strategy that is able to maintain both constraints and system consistency in the face of concurrent operations. The strategy is independent of the execution orders of concurrent operations and able to retain the effects of all operations in resolving constraint violation. In the third section, we describe the application of the proposed strategy in Collaborative Genetic Software Engineering (CoGSE) system. Comparison with related work is introduced in the fourth section and the major contributions and future work of our research are summarized in the last Section.

Constraint Maintenance

A constraint specifies a relation or condition that must be maintained in a system (Sannella *et al,* 1993, Borning *et al*, 1986). For example, resistors must obey Ohm's law. A graphic application, such as Rational Rose that represents UML concept, may specify many requirements and constraints. From the view of graphic editing systems, these requirements and constraints confine the relations or states of graphic objects.

Constraints and Constraint Satisfaction Functions

There are two kinds of constraints: static constraint, which describes a relation or condition that must be maintained at all times, and temporal constraint, which is relevant to time and event and can be described as trigger events and their responses (Borning *et al*, 1986). For example, a horizontal-line constraint is a static constraint, which requires that the y values of the both endpoints of any horizontal-line should always be equal. The description, "when object A moves, object B should be highlighted", defines a temporal constraint, which only highlights B when A moves and imposes no restriction on B when A is still.

Users' operations may violate constraints. For example, there is a constraint which defines that the color of an object, A, representing a traffic light, can only be red, green or yellow. The execution of any operation that intends to change A to other colors will violate the constraint. An operation, O, is a constraint violation operation, if its execution causes a violation of the condition or relation specified by a constraint. On the other hand, user operations may generate the events specified by a temporal constraint and trigger some responses.

A system may contain several constraints. For any constraint C_i, there is a set of Constraint Satisfaction Function (CSF), $FS_i=\{F_1, F_2,..., F_n\}$, for making that relation or condition hold. Given an operation O that violates C_i, the execution of a selected $F_i \in FS_i$ will satisfy the constraint. For example, a horizontal-line constraint, which restricts *left-endpoint.y=right-endpoint.y* of any horizontal-line, has two CSFs to satisfy it. They are (1) *left-endpoint.y←right-endpoint.y*, and (2) *right-endpoint.y←left-endpoint.y*. Function (1) means that if *right-endpoint.y*

changes, *left-endpoint.y* should be modified to the same value, and function (2) changes *right-endpoint.y* according to *left-endpoint.y* when *left-endpoint* changes its vertical position.

In graphic systems with constraints, operations may be blocked by constraint satisfaction functions. For instance, an operation that intends to color an object, representing a traffic light, to blue will be blocked. In collaborative systems, if an operation is blocked at the site where it is generated, it will not be propagated to remote sites.

On the other hand, the execution of a user operation in a graphic system may trigger some constraint satisfaction actions. For example, after a user moves *left-endpoint* of a horizontal-line, a constraint satisfaction function will be triggered to change the position of the *right-endpoint* of the line.

A Constraint Satisfaction Problem in Collaborative Systems

It is obvious that constraints are difficult to satisfy in collaborative systems regardless that they may be maintained easily in single user environments. A critical issue is that concurrent operations may result in constraint violations, which is illustrated in the following two scenarios:

Scenario 1. A horizontal-line constraint, C_1, which restricts *left-endpoint.y =right-endpoint.y* of any horizontal-line, has two CSFs to satisfy it. They are (1) *left-endpoint.y←right-endpoint.y*, and (2) *right-endpoint.y←left-endpoint.y*. Two users concurrently move both endpoints of a horizontal-line to different vertical positions.

In scenario 1, the horizontal-line constraint is difficult to maintain if both users' operations retain their display effects.

Scenario 2. Constraint C_2 confines that objects A and B should not overlap with each other and there is a CSF of C_2, which blocks any operation violating C_2. On the initial document state, A is at position P_a and B is at P_b. Two users concurrently move A and B to the same position P_c from different sites.

In scenario 2, the constraint violation is generated by concurrent operations. Even though both operations can satisfy constraint C_2 at their local sites, their executions at remote sites violate this constraint.

There is a contradiction between satisfying constraints and retaining operations' display effects in both above scenarios. The display effects of all operations cannot be retained while maintaining the constraints. The problem is caused by concurrent operations competing to satisfy the same constraint in different ways. To characterize these operations, we define competing operations group.

Definition 1. A Competing Operations Group of constraint C, denoted by COG_C, is a set of users' operations, $\{O_1, O_2, \ldots, O_n\}$, such that:

(1) For any $O_i \in COG_C$, there is $O_j \in COG_C$. O_i and O_j are concurrent,

(2) The executions of all the operations in COG_C will result in a constraint violation of C, which cannot be restored if all these operations retain their display effects,

(3) For any $O_j \in COG_C$, the executions of all the operations in $COG_C - O_j$ will not generate the condition described in (2).

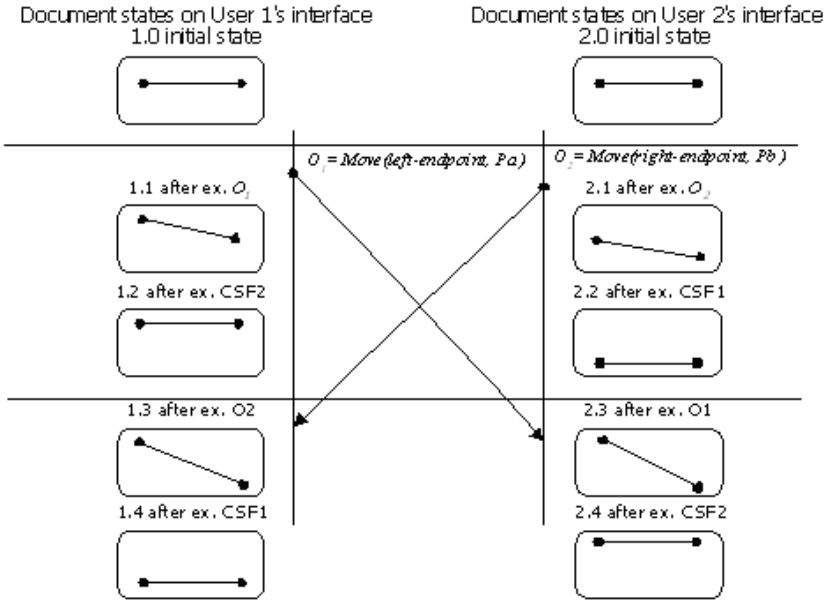

Figure 1. Maintenance of the horizontal-line constraint generates divergence

If $COG_C \neq \{\ \}$, to maintain constraint C, one operation in COG_C will lose its effect when all the operations in COG_C have been executed. If different operations lose their effects at different sites, divergence occurs, as shown in figure 1, which represents scenario 1. The document states observable from each user's interface are illustrated by rectangular boxes with rounded corners, with labels 1.0 to 1.4 for user 1 and 2.0 to 2.4 for user 2. Two concurrent operations are generated in the scenario: O_1=*Move(left-endpoint,* P_a*)* by user 1, and O_2=*Move(right-endpoint,* P_b*)* by user 2. At the site of user 1, *left-endpoint* is first moved to position P_a, resulting in the document state shown in the rectangular box 1.1. Next, constraint satisfaction function (2) of the horizontal-line constraint will be invoked to satisfy the constraint. When O_2 arrives and is executed at user 1's site, it will invoke the execution of constraint satisfaction function (1) of the horizontal-line constraint. A similar process occurs at the site of user 2. After the executions of both users' operations at each site, O_1 loses its effect at the site of user 1 and O_2 has not any effect at user 2's site. Therefore, the final results at the two sites are not identical, even though the constraint is maintained at each site.

In figure 1, operations are executed in different orders at two sites, which generate divergence. If operations can be executed in the same order at each site, both constraint satisfaction and system consistency can be achieved. Serialization undo/redo strategy ensures that operations are executed in the same order at each site according to their total ordering relation. Thus, this strategy can be adopted in collaborative systems with constraints. In scenario 2, two users concurrently execute O_1 and O_2, which move A and B to the same position P_c respectively. Suppose O_1 total ordering precedes O_2, O_2 will be blocked when it arrives at the site of user 1, because its execution at the site will violate the overlapping constraint. On the other hand, before O_1 is executed at user 2's site, O_2 will be undone. The execution of O_1 at the site of user 2 will cause O_2 to be blocked as well. Thus, both constraint and system consistency are maintained. Nevertheless, applying undo/redo strategy to satisfy constraints has many demerits. First of all, some operations may be blocked/aborted. In the above example, O_2 will be blocked at each site. Therefore, it cannot restore its effect even when O_1 is undone. Moreover, undoing/redoing a user operation may involve undoing/redoing some constraint satisfaction functions. It is complicated or may be impossible to achieve in collaborative systems with constraints. 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 constraints in collaborative systems.

It is common that concurrent operations form competing operations group in collaborative graphic applications. For instance, concurrent user operations generate cyclic Class hierarchy in a collaborative CASE system and different users concurrently connect the outputs of different circuits to the same input of a circuit in a collaborative CAD application, etc. Being able to solve this problem is crucial in the development of complex collaborative graphic systems. The challenge is that the solution should be able to maintain both constraints and system consistency. Moreover, it should be independent of the execution orders of concurrent operations.

Constraint Maintenance in Collaborative Systems

If concurrent operations form a competing operations group of constraint C, to maintain the constraint, one operation in the COG_C will have to be removed. Thus, if each site chooses the same operation in the COG_C to remove, both constraint and consistency can be maintained.

There are two methods that can be applied to remove an operation from a COG_C: blocking/aborting and masking. The blocking/aborting strategy eliminates operations' effects. Thus, if an operation is blocked/aborted, it cannot play any role afterwards. Furthermore, in collaborative graphic systems, operations may be

blocked at remote sites. This can cause confusion since the operations are not blocked immediately after generation. The alternative to blocking/aborting is masking. The masking concept is original from Multi-Version-Single-Display (MVSD) strategy in CoWord (Sun *et al*, 2004, Xia *et al*, 2004). When an object is updated by conflicting operations, multiple versions of the target object are maintained internally, but only one version is displayed at the user interface (Sun and Chen, 2002, Sun *et al*, 2004). MVSD adopts priority strategy to ensure consistency of the single displayed version at all collaborating sites. In distributed systems, an operation's timestamp can be used to represent its priority. After executing a group of conflict operations in any order, the single displayed version is the effect of the operation with the highest priority amongst all the operations in the group. Other operations are masked and their display effects are overwritten. However, even though an operation is masked, it still has a chance to recover its display effect in the future, such as when the operation with the highest priority is undone. Therefore, masking strategy retains all operations' effects. In the case of constraint violation caused by concurrency, it is usually better to preserve all users' work, rather than to destroy any user's work. Thus, we advocate using masking strategy to ensure the satisfaction of a constraint. For instance, in scenario 2, two users concurrently execute O_1 and O_2, which move objects A and B to P_c respectively. If O_2 is masked to satisfy the overlapping constraint, when O_1 is undone, O_2 may recover its display effect.

The problem of masking dealt with in constraint maintenance is more complicated than the one presented in CoWord (Sun *et al*, 2004). In CoWord, if two conflict operations are executed in the ascending order of their priorities, the operation with a lower priority will be masked automatically. Otherwise, the operation with a lower priority will be transformed to satisfy system consistency. In both cases, the operation with a lower priority is masked. However, to satisfy a constraint, concurrent operations may be masked explicitly even though they are executed in the correct sequence at each site. Concurrent operations, which cause a constraint violation, may target different objects. Hence, they cannot mask each other automatically. For example, in scenario 2, no *Move* operation can be masked automatically by the execution of the other *Move* operation. Moreover, it is difficult to apply Operational Transformation (OT) under this condition. How to mask operations to satisfy a constraint is application dependent. In the next session, we will discuss how to mask operations explicitly in a concrete collaborative graphic system.

Based on the above discussion, we can satisfy constraints by adopting a masking strategy. In scenario 1, when O_2 is ready for execution at user 1's site, a competing operations group of C_1, $COG_{C1}=\{O_1, O_2\}$, is formed. If each site masks the operation that has the biggest timestamp value in any COG_{C1} and O_1 total ordering precedes O_2, O_2 is masked at the site of user 1 and has not any effect, as shown in figure 2. On the other hand, when O_1 is ready for execution at the site of

user 2, we will obtain the same competing operations group of C_1, $COG_{C1}=\{O_1, O_2\}$. Because the timestamp value of O_2 is bigger than the timestamp value of O_1, O_2 will be masked. In the above example, the execution of O_1 at the site of user 2 will mask O_2 automatically.

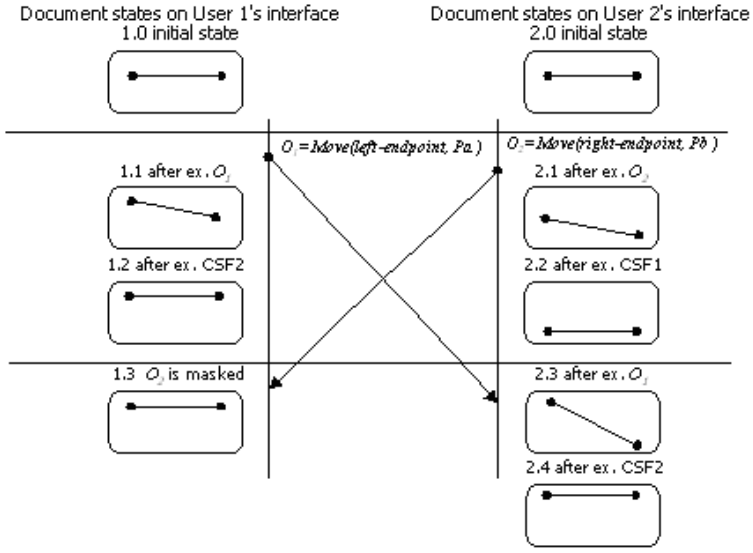

Figure 2. Maintaining the horizontal-line constraint and system consistency

At each site, O may generate many competing operations groups. We use a notation of $COGS_C{}^O=\{COG_1, COG_2, \ldots, COG_n\}$ to represent a set of COG_C of constraint C generated by applying operation O to the current document state of a site.

Definition 2. A Competing Operations Group Set of constraint C generated by operation O: $COGS_C{}^O=\{COG_1, COG_2,\ldots, COG_n\}$ is a set of competing operations groups, such that for any $COG_i \in COGS_C{}^O$, $1\le i\le n$:

(1) COG_i is a competing operations group of C,
(2) $O \in COG_i$,
(3) For any $O_j \in COG_i$, either $O_j=O$ or O_j is an executed user operation at the site and has display effect (i.e. O_j is not masked) on the current document state.

Concurrent operations may be executed in any order at each site. Therefore, an operation O may generate different competing operations group sets at different sites. Under this condition, the above masking approach may generate divergence. For example, there is a constraint C restricting that no cycle should occur in a directed-graph. Suppose the initial document contains two nodes A and B. Three users concurrently generate operations: O_1 adds a directed edge from node A to B and both O_2 and O_3 add directed edges from node B to A, as shown in figure 3-1.

Suppose our strategy is to mask the operation with the lowest priority in each COG_C to satisfy the constraint. We use the notation *O.priority* to represent the priority of operation *O*. In this example, $O_2.priority > O_1.priority > O_3.priority$.

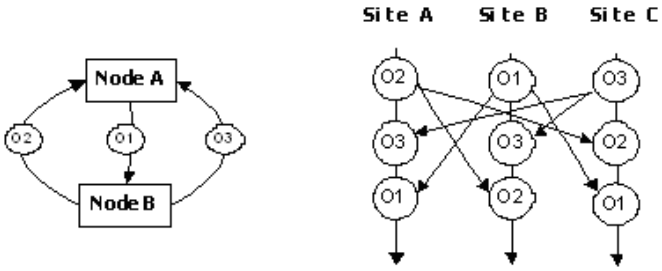

Figure 3-1. Maintenance of a non-cycle constraint

At site A, the execution order of the three operations is: O_2, O_3, O_1.

(1) O_2 is a local operation at site A, which can be executed with the constraint satisfaction on the initial document state, as shown in figure 3-2.

(2) When O_3 is ready for execution, $COGS_C^{O3}=\{\ \}$, therefore, it can be executed, as shown in figure 3-3.

(3) When O_1 arrives, its execution on the current document state will form two cycles. Thus, $COGS_C^{O1}=\{\{O_1, O_2\}, \{O_1, O_3\}\}$. Because of $O_2.priority > O_1.priority$, O_1 should be masked. $O_1.priority > O_3.priority$, then O_3 should be masked. Thus, only O_2 may have display effect. However, if O_1 is masked, the executions of O_2 and O_3 can ensure the constraint satisfaction. Therefore, only O_1 will be masked under this condition, as shown in figure 3-4.

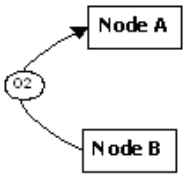

Figure 3-2. Execute O_2

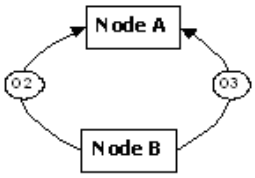

Figure 3-3. Execute O_3

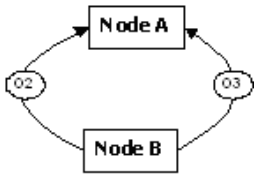

Figure 3-4. Mask O_1

At site B, the execution order of the three operations is: O_1, O_3, O_2.

(1) O_1 is a local operation at site B, which can be executed with the constraint satisfaction on the initial document state, as shown in figure 3-5.

(2) When O_3 is ready for execution, $COGS_C^{O3}=\{\{O_1, O_3\}\}$, because of $O_1.priority > O_3.priority$, O_3 is masked, as shown in figure 3-6.

(3) When O_2 is ready for execution, $COGS_C^{O2}=\{\{O_1, O_2\}\}$, because of $O_2.priority > O_1.priority$, O_1 is masked and O_2 is executed, as shown in figure 3-7.

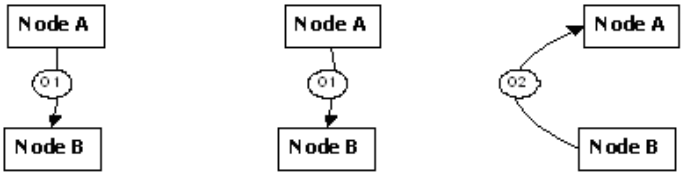

Figure 3-5. Execute O_1 Figure 3-6. Mask O_3 Figure 3-7. Mask O_1 and Execute O_2

After the three operations are executed at both sites, convergence is not maintained. The problem is that at site B, after O_1 is masked, O_3 should restore its display effect. As we mentioned previously, a masked operation may get a chance to restore its display effect when its concurrent operations are undone or masked. In the above example, when O_1 is masked, O_3 and O_2 will not generate competing operations group of C. Therefore, both of them can have display effects with the constraint satisfaction.

In collaborative systems with constraints, each time an operation O_i is masked to satisfy constraint C, all the masked operations that have lower priorities than O_i should be checked. If their executions on the new document state can satisfy constraint C, they will be unmasked. On the other hand, unmasking an operation O_j may cause other operations that have lower priorities than O_j be masked.

We can formally define the effects of the above approach. Suppose there is a constraint C and a set of operations, $OS=\{O_1, O_2,\ldots, O_n\}$, which represent all the operations that should be executed at a site. Applying the above strategy, after executing all the operations in OS (concurrent operations can be executed in any order), we will obtain the same result $R=\{O_1', O_2',\ldots, O_m'\}$. R is a set of operations, such that:

(1) $R \subseteq OS$,
(2) For any $O_k \in R$, O_k has display effect,
(3) For any $O_k \in R$, the executions of O_k and other operations in R cannot generate any COG_C,
(4) For any O_j, while $O_j \in OS$ and $O_j \notin R$, if the display effect of O_j is masked to satisfy C, the executions of O_j and other operations in R must generate COG_C and at least in one of these COG_C, O_j is the operation that has the lowest priority.

The following two procedures implement the above strategy that maintains both consistency and constraints in collaborative systems.

```
Procedure ConstraintMaintenance(O)
  {
   mGroup=GetMaskGroup(O)
   if O∈mGroup, then mask(O) and return
   else if mGroup≠{ } then for each Oi∈mGroup, mask(Oi)
   execute(O)
   if mGroup≠{ }, then {
```

```
    1) Find Os that is the operation in mGroup with the
       highest priority.
    2) In priority descending order, for any masked
       operation, Oj, if Oj.priority<Os.priority, do {
           mGroup=GetMaskGroup(Oj)
           if mGroup={ }, then unmask(Oj)
           else if Oj∉mGroup, then {
              for each Om∈mGroup, mask(Om)
              unmask(Oj)
           }
       }
  }
}
```

When an operation O that may violate constraint C is ready for execution at a site, procedure *ConstraintMaintenance* will be invoked. It calls procedure *GetMaskGroup* to obtain a group of operations to be masked to satisfy constraint C. If the group contains any operation: (1) If O is in the group, O will be masked, otherwise (2) O can be executed after all the operations in the group are masked. After an operation, O_i, is masked, all the masked operations, which have lower priorities than O_i, will be checked in the descending order of their priorities. If their executions on the new document state can satisfy constraint C, they will be unmasked.

The implementation of procedure *GetMaskGroup* is shown as follows:

```
Procedure GetMaskGroup(O)
{
  noEffectGroup={ }
  COGSo=C.getCOGS(O)
  if COGSo≠{ }, then
      for any COG in COGSo, add the operation with the
      lowest priority in the COG to noEffectGroup
  return noEffectGroup
}
```

The input of the above procedure, O, is the operation which may generate competing operations group of C at a site and the output, *noEffectGroup*, contains a group of operations to be masked to satisfy C. Function *C.getCOGS( )* obtains $COGS_C^O$ (represented as *COGSo* in the procedure). If $COGS_C^O \neq \{\ \}$, operations that have the lowest priorities in each $COG_C \in COGS_C^O$ are grouped into *noEffectGroup.* The implementation of function *C.getCOGS( )* is constraint dependent.

The above masking strategy can maintain both constraints and consistency in collaborative systems, which is independent of the execution orders of concurrent operations and able to retain the effects of all operations in resolving constraint violation. It can be adopted in many collaborative applications, including CAD, CASE, spreadsheets, graphical interface toolkits, simulation systems, etc. For instance, it may be used by:

- A graphical interface toolkit application to maintain consistency between application data and the graphic object used to display this data, when operations concurrently modify the application data and its display representation,
- An object-oriented CASE system to avoid concurrent user operations forming cyclic Class hierarchy,
- A simulation system representing current and voltage relationship of a complex circuit to confine that concurrent operations always satisfy Ohm's law,
- A graphic editing system to coordinate the concurrent operations that update graphic objects, such as coordinating the operations that concurrently change the *left, right* and *width* of a rectangle.

To illustrate the applicability of this approach, we will discuss the application of the proposed strategy in Collaborative Genetic Software Engineering (CoGSE) system in the next section.

Constraint Maintenance in Collaborative Genetic Software Engineering System (CoGSE)

Collaborative Genetic Software Engineering (CoGSE) system is a collaborative CASE (computer-aided software engineering) system based on Genetic Software Engineering (GSE). In this paper, it is used as a concrete system to demonstrate constraint maintenance in collaborative environments.

Introduction to Collaborative Genetic Software Engineering System

GSE is a methodology for software development. It uses states to characterize systems behavior and exploits the notion of state differently by making explicit use of the component-state relationship and by limiting component-component composition to a tree-form rather than a directed-graph form (Dromey, 2001, 2003). This yields an economical notation, called behavior-tree, for expressing both requirements and designs.

Using behavior-tree notation we can translate each individual functional requirement, use case, constraint or system behavior, expressed informally in natural language, into its corresponding formal graphic behavior-tree representation. Behavior trees capture/express behaviors in terms of state transitions and component interactions (Dromey, 2001). For example, expressing the behavior or requirement that "when the door is opened the light should go on" we can use a notation shown in figure 4:

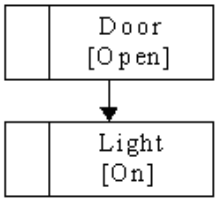

Figure 4. Door-Light requirement

Real-time Collaborative Genetic Software Engineering (CoGSE) system allows a group of users to view and edit the same behavior-tree representation at the same time from different sites. CoGSE is an Internet-based collaborative CASE system that adopts replicated architecture and is implemented in the programming language Java. The interface of CoGSE is shown in figure 5. CoGSE provides several benefits, such as easy interaction between clients and designers, sharing document easily and collaboratively integrating requirements into a design, etc.

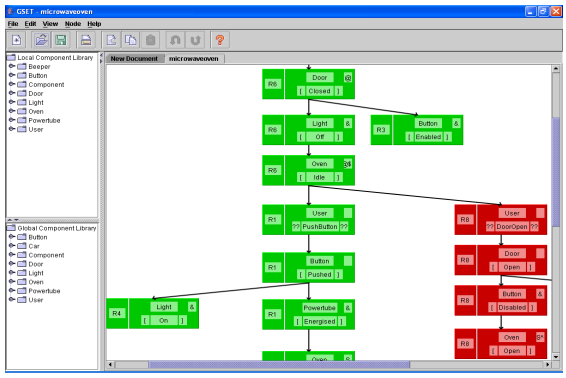

Figure 5. The Collaborative Genetic Software Engineering (CoGSE) system interface

Special Objects, Operations and Constraints in CoGSE

We use the notion of a node to describe a component in behavior-tree representation. A node can be expressed as a rectangle in graphic systems. A node, *N*, has many attributes, among which *parent* attribute denotes the parent node of *N* and *childrenList* contains all the children nodes of it. In CoGSE, each arrowed line, named edge, from a parent node to a child node represents the parent-child relation graphically. The state of an edge is dependent on the two nodes it links. In CoGSE, when a node moves or changes its size, its relevant edges will be redrawn automatically.

An *add-child* operation, *O=addChild(X, Y),* generates parent-child relation between nodes *X* and *Y*. The execution of *O* in CoGSE has three effects:

(1) Set *Y.parent=X*,

(2) Add Y to X's *childrenList*, and
(3) Create an edge from X to Y to represent the parent-child relation graphically.

Each *add-child* operation has a mask signal bit. If O=*addChild(X, Y)* is a newly arrived operation, the effect of masking O is to set mask signal bit of it. On the other hand, if O is an executed operation that has display effect, masking O means:

(1) Set mask bit of O,
(2) Delete Y from X's *childrenList*,
(3) Set *Y.parent=null*, and
(4) Delete the edge between X and Y from the graphic system.

In GSE, the behavior-tree notation is deliberately restricted to a tree like structure rather than being allowed to grow into a directed-graph (Dromey, 2001). Hence, there are two constraints, which represent the tree structure constraint, must be maintained in CoGSE:

(1) Single parent constraint C_P restricts that any node in the system cannot be pointed to by more than one node.
(2) No-cycle constraint C_N prohibits any cycle from occurring in behavior-tree representation.

Maintenance of the Tree Structure Constraint in CoGSE

The concurrent executions of *add-child* operations may violate the tree structure constraint. For example, one user executes O_1=*addChild(A, D)* and the other executes O_2=*addChild(B, D)* concurrently. Thus, C_P is violated, because node D is pointed to by both A and B. In another scenario, two users concurrently execute O_1=*addChild(A, B)* and O_2=*addChild(B, A)*. The executions of the two operations will form a cycle and violate C_N.

For any *add-child* operation O=*addChild(X, Y)* which is ready for execution at a site where the current document state is DSc, we can get $COGS_{Cp}^{O}$ of constraint C_P:

(1) If *Y.parent=null* on DSc, $COGS_{Cp}^{O}=\{\ \}$,
(2) If *Y.parent=Z* on DSc, there must be an operation O_i=*addChild(Z, Y)* which has been executed and has display effect. Therefore, $COGS_{Cp}^{O}=\{\{O, O_i\}\}$.

On the other hand, we can obtain $COGS_{Cn}^{O}$ of constraint C_N by finding all the directed paths from node Y to X on the current document state of the site. If there are n different directed paths from Y to X, after the execution of O, n cycles will be formed. Therefore, there are n competing operations groups of C_N in $COGS_{Cn}^{O}$. Each COG_{Cn} contains O and a set of operations that form a directed path from node Y to X.

In CoGSE, the maintenances of C_P and C_N may interfere with each other. For example, if three users concurrently generate operations, O_1=*addChild(A, B)*, O_2=*addChild (C, B)* and O_3=*addChild(B, C)*, the executions of these operations

will violate both C_P and C_N. Because the executions of O_1 and O_2 make B be pointed to by two nodes and O_2, O_3 form a cycle. If constraint satisfaction strategy of C_P is to mask O_2 and C_N is to mask O_3, then only O_1 will have display effect. However, if O_2 is masked to satisfy C_P, there is not any cycle anymore. Therefore, O_3 should not be masked.

The strategy to coordinate constraint maintenances of C_P and C_N is straightforward. We use a notation $COGS_{Tree}{}^{O} = COGS_{Cp}{}^{O} + COGS_{Cn}{}^{O}$ to represent all the competing operations groups of the tree structure constraint generated by applying O to the current document state of a site. If O is the operation which has the lowest priority in a $COG \in COGS_{Tree}{}^{O}$, it will be masked. As a result, other operations in each $COG \in COGS_{Tree}{}^{O}$ can be executed with the tree structure constraint satisfaction and will not be masked. On the other hand, if O is not the operation that has the lowest priority in any $COG \in COGS_{Tree}{}^{O}$, an operation in each $COG \in COGS_{Tree}{}^{O}$ will be masked to satisfy the tree structure constraint.

The following example is used to demonstrate the above approach. In figure 6, suppose the initial document contains three nodes *A, B* and *C*. Three operations, O_1=*addChild(A, C),* O_2=*addChild(B, C)* and O_3=*addChild (C, B),* are generated concurrently and $O_1.priority > O_2.priority > O_3.priority$.

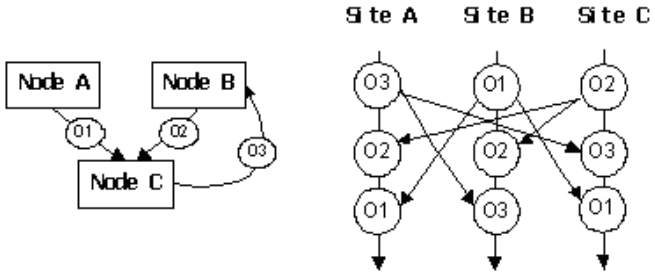

Figure 6. An example of maintaining the tree structure constraint in CoGSE

At one site, the execution order of the three operations is: O_1, O_2, O_3.

(1) The execution of O_1 on the initial document state ensures the satisfactions of both C_P and C_N, so that it can be executed directly.
(2) When O_2 arrives, it will be masked, because the executions of O_1 and O_2 violate C_P and $O_1.priority > O_2.priority$.
(3) Even though the executions of O_3 and O_2 will cause the violation of C_N. As O_2 is masked, O_3 can be executed with the satisfactions of both C_P and C_N.

Therefore, after the executions of the three operations, both O_1 and O_3 have display effects, but O_2 is masked.

At another site, the execution order of the three operations is: O_2, O_3, O_1.

(1) The execution of O_2 on the initial document state can ensure the satisfaction of the tree structure constraint, so that it can be executed directly.
(2) When O_3 arrives, it will be masked, because the executions of O_2 and O_3 violate C_N and $O_2.priority > O_3.priority$.

(3) When O_1 is ready for execution, O_2 will be masked, because the executions of O_1 and O_2 cause the violation of C_P and $O_1.priority>O_2.priority$.
(4) After O_2 is masked and O_1 is executed, O_3 will be checked. Because the executions of O_1 and O_3 do not violate the tree structure constraint, O_3 is unmasked.

Thus, after the executions of the three operations, both sites obtain the same results. The proposed masking strategy maintains both the tree structure constraint and system consistency, which is independent of the execution orders of concurrent operations.

Conflict Management in CoGSE

Different from constraints that may restrict the relations and states of many objects, a conflict is always isolated to an independent graphic object. In collaborative graphic systems, two concurrent operations, which update the same attribute of an object, conflict with each other. In this paper, we adopt the definition of conflict as below:

Conflict relation "⊗": Two *update* operations U_a and U_b conflict with each other, expressed as $U_a \otimes U_b$, if and only if: (1) they are concurrent, (2) they are targeting at the same object, and (3) they are updating the same attribute (Sun and Chen, 2002).

In collaborative graphic systems with constraints, the execution of a user operation, O, may trigger some constraint satisfaction functions whose executions will generate other operations. Hence, we can use a notation $E(O)=\{O_1, O_2,\ldots, O_n\}$ to represent the execution of a user operation O. For any $O_i \in E(O)$, either O_i is O or it is an operation generated by a triggered constraint satisfaction function. The executions of two user operations O_a and O_b will conflict with each other, if $E(O_a)=\{O_{a1}, O_{a2},\ldots, O_{an}\}$, $E(O_b)=\{O_{b1}, O_{b2},\ldots, O_{bm}\}$, there are $O_{ai} \in E(O_a)$, $O_{bj} \in E(O_b)$, and $O_{ai} \otimes O_{bj}$.

Two *add-child* operations may conflict with each other if they add edges pointed to the same node, but this kind of conflict will be solved as a result of constraint maintenance. On the other hand, *update* operations, which manipulate the graphic attributes of the graphic objects in CoGSE, may conflict with each other. For instance, two concurrent *update* operations move a node to different positions. Conflict management may interfere with constraint maintenance, even though an *add-child* operation cannot interfere with any *update* operation. For example, if conflict resolution adopts serialization undo/redo strategy, many *add-child* operations may be undone and redone to manage a conflict.

In CoGSE, Multi-Vision-Single-Display strategy (MVSD) is adopted as conflict resolution (Sun *et al*, 2004). MVSD has two distinct merits:

(1) It retains all operations' effects when conflicts occur.

(2) Different from serialization undo/redo strategy, which may impact many operations executed at a site, MVSD adopts operational transformation strategy and influences no operation except the ones involved in conflicts.

In CoGSE, constraint maintenance strategy only masks *add-child* operations, while MVSD only impacts *update* operations. Both strategies are independent of the execution orders of concurrent operations. Thus, constraint satisfaction and conflict resolution strategies will not interfere with each other.

Related Work

There is a large body of research related to constraint maintenance in graphic systems. Much work contributed to constraint maintenance in single user graphic environments (Borning *et al*, 1986, Myers, 1991, Wilde *et al*, 1990). The prevailing strategies in these systems, which implement constraint satisfaction methods intelligently to enforce system constraints, are adopted in our research. However, maintenance of constraints in concurrent environments has many new features, which cannot be handled by single user strategies.

CAB (Li *et al*, 2001), SAMS (Skaf-Molli *et al*, 2003), and CoDesign (Wang *et al*, 2002) are related to constraint control in collaborative environments. 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. However, just as its author stated that many complications in maintaining constraints in collaborative environments, such as how to handle constraint violations and coordinate interferences among constraints, are not investigated in CAB.

The intention of SAMS is to achieve semantic consistency by integrating semantic constraints to the operational transformation approach. SAMS uses the language developed in xlinkit to describe static constraints. It also discusses the questions as to where the constraints are imposed and checked and what measures should be taken in case constraints are violated. However, SAMS is based on XML resources. Its application in other environments has yet to be investigated. Moreover, it does not ensure constraint satisfaction. When a constraint is violated, it just informs users to compensate it by undoing operations or modifying objects' states.

CoDesign intends to achieve semantic preservation in real-time collaborative graphic systems and devises semantic expression to express constraints. The proposed semantic expression describes constraints as attribute and value pairs. Thus, it is only suitable for representing constraints that restrict the states of objects. It cannot represent temporal constraints and the constraints confining

relations among objects. Moreover, CoDesign also uses loose constraint satisfaction strategy that allows constraints be violated. Neither SAMS nor CoDesign investigates the relationship of constraints and no measure is taken to ensure constraint satisfaction in both systems.

By comparing with the above approaches, CoGSE adopts a novel strategy that is able to maintain both the tree structure constraint and system consistency in the face of concurrent operations. The strategy is independent of the execution orders of concurrent operations and able to retain the effects of all operations in resolving the tree structure constraint violation. Moreover, the interferences among constraints are handled in CoGSE.

Conclusion and Future Work

Constraints are very useful in CAD, CASE, and other applications. However, maintaining constraints in real-time collaborative systems is a challenging task. The difficulties are caused by concurrent operations that violate constraint(s). Being able to solve this problem is crucial in the development of complex graphic applications, such as collaborative CAD and CASE.

In this paper, we proposed a generic masking strategy to solve this problem. This solution ensures constraint satisfaction while retaining operations' effects. Our solution does not require operations to be undone/redone to achieve convergence, as undoing and redoing operations may not be possible in most systems due to the complexity of constraint satisfaction functions. We applied our solution to Collaborative Genetic Software Engineering (CoGSE) system to maintain the tree structure constraints. For these constraints, we investigated the problem of constraint interference and proposed a coordination strategy to resolve such interference.

The proposed strategy can be applied to many applications, including CAD, CASE, spreadsheets, graphical interface toolkits, simulation systems, etc. For instance, if we treat the nodes in CoGSE as Java classes in a collaborative CASE system, the proposed tree structure constraint maintenance strategy can be adopted directly to maintain Java single inheritance constraint and prevent cyclic Java Class hierarchy.

We are currently investigating conflict resolution in collaborative systems with constrains. In such systems, conflict resolutions are tightly coupled with constraint maintenances. Resolving a conflict may repair a constraint violation automatically and vice versa. This investigation may result in a more efficient method for constraint satisfaction.

Another issue we are investigating is a generic solution to coordinate interrelated constraints to achieve system consistency and constraint satisfaction. Two significant difficulties of coordinating interrelated constraints in collaborative graphic systems are:

(1) How to detect the interferences among constraint satisfaction actions? The executions of constraint satisfaction functions may interfere with each other and the interferences may propagate. Thus, the interferences among constraints may be very difficult to detect.
(2) How to coordinate constraint satisfaction actions? If some constraints interfere with each other, some strategies should be devised to determine whether all of them can be satisfied. If not, other strategies should be applied to choose operations to mask. Moreover, masking operations in multi-constraint collaborative systems may cause other interferences, which should also be coordinated.

The relationship of constraints can be very complex. If a collaborative system allows users to define constraints dynamically, there must be some strategies to coordinate the contradictory constraints, constraints' conflicts and interferences. All the above issues are currently being investigated and will be reported in subsequent publications.

References

Begole James et al. (2001): 'Resource sharing for replicated synchronous groupware', *IEEE/ACM Transactions on Networking*, vol. 9, no. 6, Dec. 2001, pp. 833-843.

Bharat, K. and Hudson, S. E. (1995): '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, pp. 121-132.

Borning, A. and Duisberg, R. (1986): 'Constraint-based tools for building user interfaces', *ACM Transactions on Graphics,* vol.5, no.4, Oct. 1986, pp. 345-374.

Chen, D. and Sun, C. (2001): 'Undoing any operation in collaborative graphics editing systems', *In Proc. of the ACM 2001 International Conference on Supporting Group Work*, Sept. 2001, pp. 197-206.

Dourish, P. (1995): 'Developing a reflective model of collaborative systems', *ACM Transactions on Computer-Human Interaction*, 2(1), Mar. 1995.

Dourish, P. (1996): '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, pp. 268-277.

Dromey, R.G. (2001): 'Genetic Software Engineering – simplifying design using requirements integration', *IEEE Working Conference on Complex and Dynamic Systems Architecture*, Dec. 2001.

Dromey, R.G. (2003): 'Using behavior trees to design large systems by requirements integration', *(Invited Paper) Dagstuhl-Seminar 03371, Scenarios: Models, Transformations and Tools, International Conference and Research Seminar for Computer Science*, Sept. 2003.

Edwards, W.K. (1997): 'Flexible conflict detection and management in collaborative applications', *In Proceedings of the ACM Symposium on User Interface Software and Technology, ACM*, New York, pp. 139-148.

Ignat, C.-L, Norrie, M.C.(2004): 'Grouping in collaborative graphical editors', *ACM Conference on Computer-Supported Cooperative Work*, Chicago, USA, Nov. 6-10, 2004, pp. 447-456.

Lafore, Robert (2003): *Data Structures and Algorithms in Java, Second Edition*, Sams Publishing, 2003.

Li, D. and Patrao, J. (2001): 'Demonstrational customization of a shared whiteboard to support user-defined semantic relationships among objects', *ACM GROUP'01*, Sept. 30-Oct. 3, 2001, Boulder, Colorado, USA, pp. 97-106.

Lin, K., Chen, D. et al. (2004): 'Tree structure maintenance in collaborative genetic software engineering system', *Proceedings of the Sixth International Workshop on Collaborative Editing Systems, ACM,* Chicago, USA, Nov. 6-10, 2004.

MacLean, A., Carter, K., Lovstrand, L. and Moran, T. (1990): 'User-tailorable systems: Pressing the issues with buttons', *In Proceedings of ACM CHI'90 Conference*, 1990.

Monfroy, E. and Castro, C. (2003): 'Basic components for constraint solver cooperations', *Proceedings of SAC*, 2003.

Myers, B. A. (1991): '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.

Sannella, M. et al. (1993): 'Multi-way versus one-way constraints in user interfaces: experience with the DeltaBlue algorithm', *SOFTWARE—PRACTICE AND EXPERIENCE*, vol. 23(5), pp. 529-566.

Skaf-Molli Hala, Molli Pascal and Ostér Gerald. (2003): 'Semantic consistency for collaborative systems', *the Fifth International Workshop on Collaborative Editing Systems Hosted by the 8th European Conference of Computer-supported Cooperative Work,* Helsinki, Sept. 15, 2003.

Sun, C., et al. (1998): 'Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems', *ACM Transactions on Computer-human Interaction*, 5(1), Mar. 1998, pp. 63-108.

Sun, C. and Chen, D. (2002): 'Consistency maintenance in real-time collaborative graphics editing systems', *ACM Transactions on Computer-Human Interaction*, vol. 9, no.1, Mar. 2002, pp. 1-41.

Sun, C. (2002): 'Undo as concurrent inverse in group editors', *ACM Transactions on Computer-human Interaction,* vol. 9, no. 4, Dec. 2002, pp. 309-361.

Sun, D. et al. (2004): 'Operational transformation for collaborative word processing', *ACM Conference on CSCW*, Chicago, USA, Nov. 6-10, 2004.

Valiente Gabriel (2002): *Algorithms on Trees and Graphs,* Springer Verlag, Nov. 1, 2002.

Wang, X.Y, Bu J.J and Chen, C. (2002): 'Semantic preservation in real-time collaborative graphics designing systems', *The Fourth International Workshop on Collaborative*, New Orleans, USA, 2002.

Wilde, N. and Lewis, C. (1990): 'Spreadsheet-based interactive graphics: from prototype to tool', *In Proceedings of ACM CHI'90 Conference on Human Factors in Computing Systems*, Application Areas, 1990. pp. 153-159.

Xia, Q. et al. (2004): 'Leveraging single-user applications for multi-user collaboration: the CoWord approach', *ACM Conference on CSCW*, Chicago, USA, Nov. 6-10, 2004, pp. 162-171.