Multi-way Dataflow Constraint Propagation in Real-time Collaborative Systems

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

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

***Abstract*—Constraints are very useful in real-time collaborative editing systems. They are able to automatically enforce semantic rules and properties. A specific type of constraint is dataflow constraint. Any property that can be expressed as an equation can be represented as a dataflow constraint. However, ensuring multi-way dataflow constraint satisfaction and consistency maintenance in a replicated collaborative environment is a challenge. This paper presents a novel method for computing multi-way dataflow constraint propagation for real-time collaborative editing systems. This method produces convergent result that is consistent with syntax level effect, irrespective of the operation execution order. This method is generic and is applied to enforce object placement and label name consistency in a real-time collaborative CASE system.**

Keywords-multi-way dataflow constraint; constraint propagation; real-time collaborative systems

I. INTRODUCTION

Constraints specify semantic level conditions that must be satisfied, and will automatically be maintained by the constraint system. For instance, they may be adopted by a spreadsheet system to denote the relationship of different cells, a graphical system to specify the positions of graphic objects, etc. Constraint-based applications simplify users' jobs by allowing users to concentrate on saying what should be true, leaving it to the constraint systems to worry about when and how to make these things true [3].

This paper concentrates on a frequently used type of constraint called dataflow constraint that is capable of expressing relationships over multiple data types and is conceptually simple [2], [8]. Any requirement that can be expressed as an equation can be represented as a dataflow constraint. For instance, the requirement defining "point *A* should be in the middle of the line connecting points *B* and *C*" can be reduced to a dataflow constraint "$A=(B+C)/2$".

A major issue that needs to be solved when developing dataflow constraint systems is being able to propagate update effects in order to maintain the constraint. For instance, in a graphical system, constraint propagation can be used to maintain constraints between graphical objects when they are moved. If object *B* is kept to the right of *A*, expressed as "*B.left=A.left+A.width*", and the end-user moves object *A* sideways, then *B* will be moved with it as a result of constraint propagation.

New challenges arise when adopting dataflow constraints in real-time collaborative systems. Firstly, operation execution and constraint propagation effects need to be consistent with the underlying syntax level execution effect. Secondly, due to replicated nature of the systems, convergent propagation effects need to be ensured at all replica sites. This has to be achieved under the condition that concurrent actions may be generated to update variables in the same constraint. Thirdly, as constraint propagation may take arbitrary amount of time to compute, the ability to allow locally generated operations to be executed before operation propagation is required. This is to ensure fast local response time.

In this paper, we proposed an efficient constraint propagation method which is able to maintain both dataflow constraints and consistency in real-time collaborative systems.

II. 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 [8]. Each CSM uses some of the constraint's variables as inputs and computes the remainder as outputs [2]. For example, suppose constraint *C* defining "$X=Y+Z$" is associated with a CSM, "$X\leftarrow Y+Z$", which means that *X* should be calculated according to *Y* and *Z*. Each time a user updates either *Y* or *Z*, the constraint system will invoke the CSM updating *X* accordingly to satisfy the constraint.

A dataflow constraint could be one-way or multi-way. If a constraint has exactly one CSM that is used to satisfy it, it is a one-way constraint. On the other hand, a multi-way constraint has multiple CSMs that can be used to satisfy it. Multi-way dataflow constraints can express relationships in multiple directions and have a number of advantages over one-way ones [2], [6], [8].

A CSM may have only one output or multiple outputs. Obviously, multi-output constraints are more expressive than single-output ones. However, multi-way, multi-output constraints have drawbacks which impeded their acceptance. It is proved that satisfaction of multi-way, multi-output constraint is NP-complete [8]. Moreover, the constraint satisfaction results of multi-way, multi-output constraints are unpredictable [8].

In this paper, we focus on multi-way, single-output constraints. In addition, a multi-way constraint has a CSM for calculating a value for each of the variables it constrains, in terms of the values of the other variables [8]. Each CSM uses all the variables confined by the constraint, one as the output and the others as the inputs. For instance, C, defining "$X=Y+Z$", is associated with three CSMs, "$X\leftarrow Y+Z$", "$Y\leftarrow X-Z$" and "$Z\leftarrow X-Y$".

Dataflow constraints are commonly expressed in terms of constraint graphs. Initially, the constraint system is represented as an undirected bipartite graph [2], [6], [8], such as Fig. 1 that represents two constraints. C_a defines "$W=X+U$" and C_b constrains "$X=Y+Z$". Here, a circle represents a variable and a square expresses a constraint. A set of undirected edges denotes the relationship between variables and constraints.

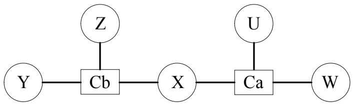

Figure 1. Graphic representations of constraints

If a CSM, f, is selected to satisfy constraint C, all the inputs to f are represented as directed edges from the input variables to C and a directed edge from C points to f's output. In Fig. 2, CSM "$X\leftarrow Y+Z$" is used to satisfy C_b. Here, Y and Z are the inputs and X is the output of C_b.

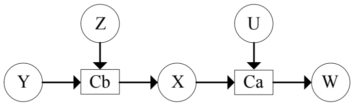

Figure 2. A solution graph

A set of constraints, $CS=\{C_1, C_2, \ldots, C_n\}$, is satisfiable if for each $C_i \in CS$, $1 \le i \le n$, a CSM 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. A direct graph that satisfies these two conditions is called a solution graph which represents a computation flow to satisfy a set of constraints [8].

III. A CONSTRAINT PROPAGATION METHOD

In constraint-based interactive systems, users may update constrained variables, which causes constraint violations. Constraint propagation provides an efficient way to re-satisfy constraints.

Propagation, which is a generalization of data-driven computation, works very effectively in interactive systems. Consider the constraint defining "*left-endpoint.y=right-endpoint.y*" of a horizontal-line. Any assignment to the variable *left-endpoint.y* causes an assignment to *right-endpoint.y*. Here, the change of *left-endpoint.y* is propagated to *right-endpoint.y*. The constraint maintenance is achieved by taking user operations as inputs, performing propagation, and outputting the consequences.

When operation O assigns constrained variable V a new value, constraint propagations should be performed for all the constraints associated with V to satisfy them. In general, each variable may be involved in many constraints. Consequently, the assignment of a new value to a given variable as a result of propagation may propagate further new assignments to other variables, which may cause further propagation in their turn.

Determining the propagation path for an *Update* operation is a critical issue. *U(object.key, (new-value, new-priority), (old-value, old-priority))* denotes an *Update* operation which updates the attribute *key* of *object* from o*ld-value* to *new-value* [7]. According to the Priority Assignment Scheme (PAS) introduced in [7], when a user generates an *Update* at a site, its *new-value* parameter shall be assigned with the current highest priority available, and its *old-value* shall be assigned with the lowest priority. We refer the priority assigned to the *new-value* parameter of an *Update* as the priority of the operation. As the priorities assigned to different *Updates* are totally ordered [7], we use a sequence of positive integers to represent the totally ordered priorities in this paper for the sake of conciseness.

To propagate the effect of O updating constrained variable V, we may build an arbitrary solution graph, where V is not the output of any constraint (As V is determined by O rather than by any constraint). Then re-satisfy each constraint according to the solution graph. However, to construct a solution graph, we should examine the entire constraint set. A change to a constrained variable usually perturbs only a portion of constraints, so that it is more expedient to determine propagation path incrementally based on the previous propagation result.

Given operation O updates constrained variable V on document state So where all the constraints are satisfied. Let Go be the solution graph denoting the computation flow to satisfy all the constraints on So. According to the definition of solution graph, V could be the output of at most one constraint and an input of some constraints in Go, as shown in Fig. 3. Here, a directed dashed line connecting two variables indicates that there may be many variables and constraints in the directed path between the two variables.

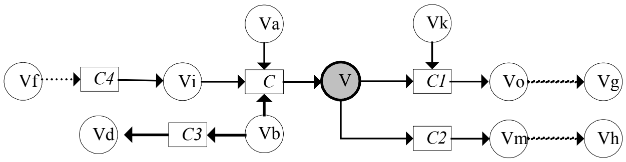

Figure 3. The initial solution graph of a system

For any constraint C_i, one of whose inputs is V in Go, such as C_1 and C_2 in Fig. 3, constraint propagation should be performed to satisfy it after V is updated. Let V_o be the output of C_i in Go. It is desirable that the change of V is propagated to V_o, because (1) constraint propagation can be conducted according to the propagation path defined in Go, so that constructing new propagation path is unnecessary, and (2) V_o is the output of a constraint in Go, so that it is not determined by a user operation on document state So. The propagation result will not mask the effect of any user operation. For the same reason, when V_o is updated as a result of constraint propagation, the change of V_o should be propagated to the output of any constraint, one of whose inputs is V_o in *Go*. Consequently, the change of V should be propagated to all the downstream variables of V in *Go*.

On the other hand, the change of V may also be propagated to some of its upstream variables in *Go*. V is the output of constraint C in *Go*, as shown in Fig. 3. When V is updated by operation O, it is determined by O rather than by C. Therefore, the computation flow satisfying C in *Go*, where V is the output of C, cannot be applied. To re-satisfy C, the change of V should be propagated to another constrained variable of C. V becomes an input of C and another C's variable should be changed to the output of C. In Fig. 3, any one of V_a, V_b and V_i could be the new output of C. Given V_i becomes the new output of C. As V_i is the output of C_4 in *Go*, when it becomes the output of C, it should be changed as an input of C_4 (a variable cannot be the output of more than one constraint in a solution graph). Consequently, one of the inputs of C_4 should be assigned as the new output. The upstream propagation continues until reaching variable V_f which is not the output of any constraint in *Go*, as shown in Fig. 4. Accordingly, the change of V is propagated to every one of the downstream variables of V in figure Fig. 4.

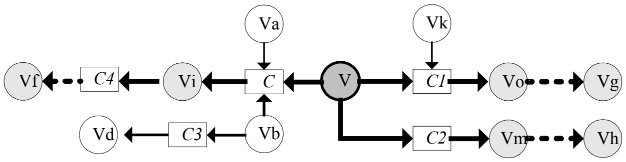

Figure 4. The solution graph after constraint propagation

According to the above discussion, if n variables are the upstream variables of V but not the output of any constraint in *Go*, we may construct n different solution graphs to conduct the constraint propagation caused by O. Our proposed scheme ensures that all the collaborating sites produce the same final solution graph.

The key idea behind our constraint-propagation-method is to associate sufficient information with each variable to enable the method to determine propagation paths: The value of variable V is associated with a priority, expressed as *V.value.priority* and referred as the priority of V. When V is updated by O, set *V.value=O.new-value* so that *V.value.priority =O.new-value.priority*. If V is the output of constraint C, $V.value.priority=V_i.value.priority$, while V_i is the variable with the lowest priority among all the inputs of C.

V is also associated with a level property, denoted as *V.level*. If V is not the output of any constraint in a system, *V.level=0*. Otherwise, if V is the output of constraint C, then $V.level=V_i.level+1$, while V_i is the variable with the lowest priority among all the inputs of C.

We define the power of variable V as a tuple, (*V.value.priority, V.level*), denoted as *V.power*. For any two constrained variables V and V_i, the power of V is lower than the power of V_i, denoted as $V.power<V_i.power$, if and only if (1) $V.value.priority<V_i.value.priority$, or (2) $V.value.priority=V_i.value.priority$ and $V.level>V_i.level$.

The powers of constrained variables can be used to guide the directions of constraint propagations. In our method, when constrained variable V is updated, for any constraint C that is associated with V, the change should always be propagated to the constrained variable which has the lowest power among all of C's variables.

Function constraintPropagation() is invoked to perform constraint propagation each time a constrained variable is assigned a new value, which is sketched below.

```
Procedure constraintPropagation(V,C)
{
For any constraint Ci associated with V while Ci≠C,
    Vo=getLowestPowerVariable(Ci)
    call Vo←f(V, V1, ..,Vn)
    Vi=getLowestPowerVariable(f().getInputs())
    Vo.value.priority=Vi.value.prirority
    Vo.level=Vi.level+1
    call constraintPropagation(Vo, Ci)
}
```

The input parameter, V, of the above function is a variable which is assigned a new value by an *Update* or a CSM. C is a constraint associated with V and constraint propagation has been performed for it after V is updated. When an *Update* assigns V a new value, V and *null* will be passed as inputs to the function. Accordingly, constraint propagation will be performed for any constraint C_i associated with V. Method invoking getLowestPowerVariable(C_i) returns constrained variable V_o, which has the lowest power among all the constrained variables of C_i. $V_o \leftarrow f(V, V_1, ..V_n)$ denotes the CSM associated with C_i, whose output is V_o. Method invoking getLowestPowerVariable(f().getInputs()) returns variable V_i which has the lowest power among all the inputs of the CSM. Accordingly, $V_o.value.priority=V_i.value.prirority$ and $V_o.level=V_i.level+1$. If V_o is also associated with other constraints, after V_o is updated, constraint propagations will be performed for these constraints by recursively invoking the function.

We have proved that the proposed method is able to maintain both constraint and consistency in collaborative systems, which is independent of the execution orders of concurrent operations.

IV. OPTIMIZATION

In section III, the analysis is under the situation that constraint propagation is performed immediately after each user operation that updates a constrained variable. Suppose m operations update constrained variables in a system with n constraints. In the worst case, each operation may cause constraint propagation for n constraints, to satisfy these constraints we should perform $m \times n$ times constraint propagations. Thus, the time complexity is $O(n^2)$.

Performing constraint propagation each time a constrained variable is assigned a new value is unnecessary. For instance, when two users concurrently update the positions of *left-point* and *right-point*, to satisfy constraint C_p, defining *"middle-point=(left-point+right-point)/2"*, we can perform constraint propagation only once, which changes the position of *middle-point* by taking into account the effects of both user operations.

To improve system-responsiveness, if any user operation is waiting for execution, constraint propagation will not be performed. Each site maintains a Constraint-Propagation-Buffer (CPB), which is to record constraints whose constrained variables have been updated and constraint propagations should be performed to satisfy them. Each time a constrained variable of *C* is assigned a new value at a site, constraint *C* will be recorded in the CPB of the site.

In the best case, constraint propagation is performed after all the *m* operations have been executed. Obviously, at most all the *n* constraints are recorded in CPB after the *m* operations have been executed. If we know the final solution graph *Gn* in advance, the most efficient way to conduct constraint propagation is to satisfy *C* before all of its downstream constraints in *Gn*, which is the strategy used to satisfy constraints on the initial document state. Thus, constraint propagation will be performed for each constraint only once.

Even though we cannot predict the final solution graph *Gn*, we know that if C_a is an upstream constraint of C_b in *Gn*, its output must have a higher power than the output of C_b. Moreover, the power of the output of C_a in *Gn* will be set according to the power of the constrained variable with the second lowest power among all the constrained variables of C_a. Therefore, we sort the constraints in the CPB. C_a is ordered before C_b in a CPB, if and only if $V_1.power>V_2.power$, while V_1 is the constrained variable of C_a whose power is the second lowest among all the constrained variables of C_a, and the same is for V_2 to C_b. Performing propagations for constraints recorded in CPB in sequence, the time complexity of the proposed schema is *O(n)* in the best case.

V. SYSTEM STRUCTURE

The structure of a constraint-based collaborative system is shown in Fig. 5.

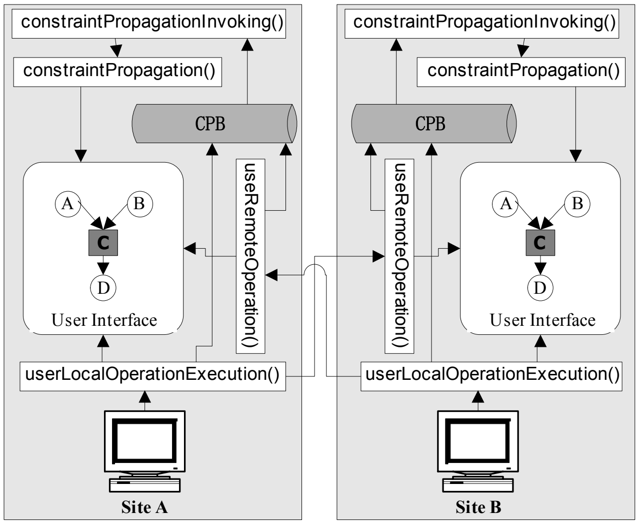

Figure 5. System structure

Any user operation can be executed immediately at the site it was generated, even if there are remote user operations waiting for executions. Each site maintains a function, userLocalOperationExecution(), which is invoked to execute a local *Update*, as described below:

```
//Ua is a local user operation updating V
userLocalOperationExecution(Ua)
{
 execute Ua
 if V is a constrained variable, then
        V.level=0
        for every constraint C associated with V
             record C in CPB
         reorder the constraints in CPB
}
```

When an *Update* is executed at the site it was generated, it is dependent on all the operations that have been executed at the site, so that it will not be transformed against any operation. If the operation updates a constrained variable, the level of the variable is set to *0*, as the variable is not the output of any constraint after the execution of the *Update*. Moreover, all the constraints associated with the variable will be recorded in CPB.

To execute remote *Updates*, each site maintains a function, userRemoteOperationExecution(), which is described below:

```
//Ua is a remote user operation updating V
userRemoteOperationExecution(Ua)
{
if V is not a constrained variable, then
according to operation-execution order, for any executed
operation Ub conflicting with Ua,
     Ua=conflictResolution(Ua, Ub) [7]
     execute Ua
else  // if V is a constrained variable
     original=V.value.priority
     if (Ua.new-value.priority<V.value.priority)
          Ua.new-value=V.value
     Ua.old-value=V.value
     execute Ua
     if original≠V.value.priority, then
          V.level=0
          for every constraint C associated with V
               record C in CPB
          reorder the constraints in CPB
}
```

When user operation U_a updating *V* is ready for execution at a remote site, if *V* is not associated with any constraint, U_a will be transformed against all of its conflicting operations that have been executed at the site so that the transformed operation can achieve the correct effects and maintain document consistency [7]. The execution of the transformed U_a will not invoke any constraint propagation. On the other hand, if *V* is a constrained variable, U_a can have effect on the current document state and cause constraint propagation only if $U_a.new\text{-}value.priority>V.value.priority$. If U_a assigns *V* with a new value, all the constraints associated with *V* will be recorded in CPB.

Function constraintPropagationInvoking() will be invoked at each site when the system starts up. The function keeps running in the background. If no user operation is waiting for execution at that site, this function will fetch constraints from the local CPB and send them in sequence to function constraintPropagation() to perform constraint propagation.

```
constraintPropagationInvoking()
{
  while (true)
    if no user operation is waiting for execution  and CPB
contains any constraint, then
        get the first constraint, C, in CPB
        call  constraintPropagation(C)
        delete C from CPB
}
```

Function constraintPropagation() is maintained at each collaborating site, which is sketched below.

```
constraintPropagation (C)
{
Vo=C.getLowestPowerVariable()
call Vo←f(V, V1, ..,Vn)
Vi=getLowestPowerVariable(f().getInputs())
Vo.value.priority=Vi.value.prirority
Vo.level=Vi.level+1
for any Ci, while Ci≠C and Vo is a constrained variable of Ci,
record Ci in CPB
reorder the constraints in CPB
}
```

The input parameter, *C*, of function constraint-Propagation() is a constraint whose constrained variables have been updated, and constraint propagation should be performed to satisfied it. Method C.getLowestPowerVariable() returns a constrained variable V_o, which has the lowest power among all the constrained variables of *C*. $V_o \leftarrow f(V_1, V_2, .., V_n)$ denotes the CSM associated with *C*, whose output is V_o and inputs are all the other constrained variables of *C*. Method getLowestPowerVariable(f.getInputs()) returns constrained variable V_i which has the lowest power among all the inputs of the CSM. Accordingly, $V_o.value.priority=V_i.value.prirority$ and $V_o.level=V_i.level+1$. If V_o is also associated with other constraints, constraint propagations should be performed for these constraints after V_o is updated. Therefore, these constraints should be recorded in CPB.

VI. CONCLUSION AND FUTURE WORK

Multi-way dataflow constraints are useful in single user editing systems, and even more useful in real-time collaborative systems. However, building a collaborative system that supports such constraints is a major challenge.

In this paper, we have presented a novel constraint propagation method, to maintain multi-way single-output dataflow constraints in real-time collaborative environments. Consistency of propagation effect is maintained at all replica sites while allowing operations to be executed in any order. Compare with the method introduced in [5], this method is more advanced. It is able to produce propagation effect that is consistent with the underlying syntax level execution effect. Furthermore, constraint propagations are performed only when no user operation is waiting for execution. This improves system-responsiveness.

The method we have presented can be applied to many kinds of collaborative applications, including collaborative CAD, CASE, spreadsheets, graphic editing systems, etc. We have chosen to implement this method in our Collaborative Genetic Software Engineering System (CoGSE). CoGSE is a collaborative CASE system that allows multiple users to draw Behavior Tree diagrams to represent the behavior of software systems [1], [4]. One of the constraints that are implemented is to ensure objects of the same level line up horizontally. Another constraint is to ensure if a label is changed, then all the labels with the same name will automatically be updated.

There are some limitations in applying our method. It is only applicable to equality, not inequality constraints. Furthermore, it is designed to maintain predefined constraints. If constraints are added/deleted dynamically, the method cannot ensure system consistency. The solutions to these problems are currently being investigated, and will be reported in our subsequent publications.

Over the last fifteen years, real-time collaborative systems have moved from being prototypes in laboratories to becoming usable commercial systems and also freeware. So far, much of the research and development has concentrated on syntax level consistency. With the use of constraints in supporting application level semantics, we hope to make real-time collaboration even more productive and easier to use.

REFERENCES

[1] R.G. Dromey, “Using behavior trees to design large systems by requirements integration”, Scenarios: Models, Transformations and Tools, International Conference and Research Seminar for Computer Science, Sep. 2003.

[2] BN Freeman-Benson, J. Maloney, and A. Borning., “An incremental constraint solver”, Communications of the ACM, vol.33, no.1, pp.54-63, Jan. 1990.

[3] D.R. Hill, “The RENDEZVOUS constraint maintenance system”, In Proceedings of the ACM Symposium on User Interface Software and Technology, pp.225-234, 1993.

[4] K. Lin, D. Chen, C. Sun and R.G. Dromey, “Maintaining constraints in collaborative graphic systems: the CoGSE approach”, In Proceedings of the 9th European Conference on Computer Supported Cooperative Work, Paris, 2005.

[5] K. Lin, D. Chen, C. Sun and R.G. Dromey, “Maintaining multi-way dataflow constraints in collaborative systems”, In Proceedings of IEEE 2005 International Conference in Collaborative Computing: Networking, Applications and Worksharing, San Jose, CA, USA, Dec. 2005.

[6] M. Sannella, J. Maloney, BN Freeman-Benson, and A. Borning “Multi-way versus one-way constraints in user interfaces: experience with the DeltaBlue algorithm”, Software-Practice and Experience, vol.23, no.5, pp.529-566, 1993.

[7] D. Sun, S. Xia, C. Sun, and D. Chen , “Operational transformation for collaborative word processing”, In Proceedings of the ACM Conference on Computer Supported Cooperative Work, Chicago, USA, pp.437-446, Nov. 6-10, 2004.

[8] B. Zanden, “An incremental algorithm for satisfying hierarchies of multi-way dataflow constraints”, ACM Transaction on Programming Languages and Systems, vol.18, no.1, pp.30-72, Jan.1996.