Properties of Constraint Systems of Property Models

John Freeman
Texas A&M University
jfreeman@cse.tamu.edu

Jaakko Järvi
Texas A&M University
jarvi@cse.tamu.edu

Jacob Smith
Texas A&M University
jnsmith@cse.tamu.edu

Mat Marcus
mmarcus@emarcus.org

Sean Parent
Adobe Systems Inc.
sparent@adobe.com

Abstract

This report describes some properties of a class of multi-way dataflow constraint systems with hierarchies. This class arises in realizing *property models*, an approach that enables a high level of reuse in programming user interfaces. We prove that a constraint system in this class has a unique solution if one exists; define a class of changes in a constraint hierarchy that are guaranteed to keep the solution of the constraint system unchanged; and determine the extent to which other changes in a constraint hierarchy affect the solution of the constraint system.

1. Constraint Systems for Property Models

Constraint systems, in particular *multi-way dataflow constraint systems* [7], are useful for representing property models [3]. A constraint system consists of *variables* and *constraints*. Abstractly, constraints are defined as relations between subsets of those variables; *solving* the constraint system means finding a valuation for the system's variables so that all relations in the system are satisfied. Below we give a detailed description of the constraint system for representing property models, a particular form of a multi-way dataflow constraint system with hierarchical constraints, and how it can be efficiently solved. The constraint system has a natural representation as a particular kind of graph; we describe this representation as well.

1.1 Multi-way dataflow constraint systems

A constraint system S is a tuple $\langle V, C\rangle$, where V is a set of variables, each having a *current value*, and C is a set of constraints. Each constraint in C is a tuple $\langle R, r, M\rangle$, where $R \subseteq V$, r is some n-ary relation between variables in R ($n = |R|$), and M is a set of *constraint satisfaction methods*, or just *methods*. If the values of variables in R satisfy r, we say that the constraint is *satisfied*. Executing any method m in M *enforces* the constraint by computing values for some subset of R, using another disjoint subset of R as inputs, such that the relation r becomes satisfied. We refer to the input and output variables of a method m as $ins(m)$ and $outs(m)$, respectively. The code realizing a method is considered a "black box"—it is the programmer's responsibility to ensure that a constraint is satisfied after any of its methods is executed.

The relations in constraint systems can be arbitrary, but often they are equalities. For example, the equation $r = w/h$, name it e_1, could describe what should hold true of the width w, height h, and aspect ratio r of an image. One possible set of methods for this equation is $M_1 = \{w \leftarrow rh, h \leftarrow w/r, r \leftarrow w/h\}$. Another relation, e_2, might connect the width and height to the size of the file necessary to represent the image, say, $s = g(wh)$. We assume that the function g realizes the size calculation, and that its details are not known. For the relation e_2, a possible (singleton) set of methods is $M_2 = \{s \leftarrow g(wh)\}$; that is, the constraint can only be satisfied in one direction, by computing s from w and h.

The constraint satisfaction problem for a constraint system $S = \langle V, C\rangle$ is to find a valuation of the variables in V such that each constraint in C is satisfied. Such a valuation is attained if a set of methods, exactly one method from each constraint in C, is executed in an order where once a variable has been read from or written to by one method, no other method will write to it. A solution is thus characterized by a sequence of methods, often called a *plan*. Depending on the problem, such a plan may or may not exist, and may or may not be unique. If a plan exists for a given constraint system, we say the system is *satisfiable*; otherwise it is *over-constrained*. If more than one plan exists, the system is *under-constrained*. For over-constrained systems, variations of the constraint satisfaction problem use different criteria to relax some constraints in order to find a plan that can satisfy a subset of the constraints [7, 2, 4]. In particular, constraint hierarchies [1] classify constraints based on their *strength*; weaker constraints can be retracted in order to satisfy stronger ones.

We place four well-formedness conditions on the constraint systems for property models. All of them are easily checked, so that models violating the conditions can be rejected, and easily satisfied, so that they do not significantly restrict what programmers can express.

Definition 1 (Well-formed constraint system)**.** A constraint system $S = \langle V, C\rangle$ is *well-formed* if

(WF-1) for all constraints $\langle R, r, M\rangle$ in C, for all methods m in M, $ins(m) \sqcup outs(m) = R$;[1]
(WF-2) for all methods m in S, $outs(m) \neq \emptyset$;
(WF-3) for any two constraints $\langle R_1, r_1, M_1\rangle$ and $\langle R_2, r_2, M_2\rangle$ in C, $R_1 \neq R_2$; and

[1] $\sqcup$ denotes the disjoint union.

(WF-4) for all constraints $\langle R, r, M\rangle$ in C, for any two methods m_1, m_2 in M, $outs(m_1) \not\subseteq outs(m_2)$.

The condition WF-1 is known as *method restriction* [5, p. 56], and it guarantees that a multi-way constraint system can be solved in polynomial time [6] with respect to the number constraints. Constraints that contain a method that violates WF-2 are unnecessary; such constraints can always be satisfied with no effect on the valuation of the system's variables. WF-3 disallows two constraints to define a relation between the exact same set of variables; no system violating WF-3 (but respecting WF-1 and WF-2) is satisfiable. WF-4 rules out methods that would never be selected as part of a plan.

1.2 Constraint graph

A well-formed multi-way dataflow constraint system is in a one-to-one correspondence with an *oriented*, *bipartite* graph $G = \langle V + M, E\rangle$, with vertex sets V and M representing the variables and methods of the system, respectively, and E the directed edges that connect each method to its input and output variables. Where $v, u \in V$ and $m \in M$, the edge (v, m) indicates that the variable v is an input of the method m, and (m, u) that m outputs to the variable u. The values of variables are labels of the vertices in V and the code of the methods the labels of the vertices in M. The graph is oriented, that is, $(a, b) \in E \implies (b, a) \notin E$, because for each method m, $ins(m)$ and $outs(m)$ are disjoint. Below, we overload the ins and $outs$ functions for graph vertices as well.

The grouping of methods and variables into constraints is not explicit in the representation $G = \langle V + M, E\rangle$. One way to add this information is as *hyperedges* that connect subsets of M. These hyperedges are, however, uniquely determined by G alone. To explain, we use the notion of a *neighborhood* (nbh) of a vertex: in a graph $\langle V, E\rangle$, $nbh(a) = \{b \mid (a, b) \in E \text{ or } (b, a) \in E\}$. Now let $\sim_{nbh}$ be the equivalence kernel of nbh, defined by $m_1 \sim_{nbh} m_2 \Leftrightarrow nbh(m_1) = nbh(m_2)$. Assuming method restriction (WF-1), all method vertices of the same constraint belong to the same equivalence class in the quotient set $M/{\sim_{nbh}}$. Furthermore, by WF-3, $[m_1]_{nbh} \neq [m_2]_{nbh}$ implies that m_1 and m_2 are methods from two different constraints. Therefore, two methods m_1 and m_2 belong to the same constraint if and only if $[m_1]_{nbh} = [m_2]_{nbh}$. In the following, we omit the subscript $\cdot_{nbh}$ and just write $[\cdot]$ and $\sim$. The elements of the quotient set $M/{\sim}$ can thus be identified with the hyperedges representing the constraints of G; the labels of these hyperedges represent the constraint strengths.

1.3 Solution graph

A solution of a constraint system can be explicitly represented as a subgraph of the constraint graph, called a *solution graph*. We use the notation $G[V]$ to indicate the *vertex-induced* subgraph of G. If V is a subset of G's vertex set, $G[V]$ is the graph whose vertex set is V and the edge set includes all edges of G whose both endpoints are in V. We denote the number of in-edges of a vertex v in the graph G as $d^-_G(v)$, and the number of out-edges as $d^+_G(v)$.

Definition 2 (Solution graph)**.** Let $G = \langle V + M, E\rangle$ be a constraint graph. Let $M' \subseteq M$. $G' = G[V + M']$ is a *solution graph* of G iff (1) G' is *acyclic*, (2) $\{[m] \mid m \in M'\} = M/{\sim}$, (3) $|M'| = |M/{\sim}|$, and (4) $\forall v \in V. d^-_{G'}(v) \leq 1$.

The second and third conditions together establish that M' contains exactly one method from each constraint, and the fourth that no two methods output to the same variable. The third condition is implied by the first and the fourth because of WF-1.

The set of method vertices in a solution graph is a plan of a constraint system. Executing the methods in the plan according to a topological ordering of the vertices of the solution graph will give the system's variables a valuation that satisfies all its constraints.

1.4 Constraint hierarchies

The constraint systems arising in property models are usually under-constrained initially. To deal with an under-constrained constraint system, we add so called *stay constraints* [1] to the system. A stay constraint consists of a single method, which we call the *stay method*, with one output and no inputs. The method is a constant function that has the current value of its output variable. Thus, if a method of a stay constraint is executed, the value of the variable *stays* the same.

In our approach, every variable is given a stay constraint. As a result, the system becomes over-constrained and is no longer satisfiable.[2] To deal with the over-constrained systems, we follow the approach of constraint hierarchies [1], where each constraint in the system is assigned a *strength*, and the solution of the over-constrained system is defined as the solution to the "best" satisfiable constraint system that retracts some of the system's constraints. Intuitively, the best system is the one that retracts the weakest and fewest constraints. Usually certain constraints are assigned a special strength, we call it *must*, which indicates that no solution can retract those constraints. An *admissible solution* is one that enforces all constraints with the strength *must*.

Many possible criteria for what "better" means have been used in different kinds of constraint systems. A suitable starting point for property models is the "locally-predicate-better" criterion [2]: if one solution enforces a constraint that the other does not, and every weaker constraint is either retracted in both solutions or enforced in both solutions, then the former solution is locally-predicate-better than the latter. This comparator deems two solutions incomparable if, moving from stronger to weaker strengths, the two solutions enforce different subsets of constraints from the same level of the hierarchy. Our formulation considers the solution that enforces a larger number of constraints in a set of equal-strength constraints better. This difference is inconsequential when the strengths of all retractable constraints are unique, as is the case with property models.

Formally, we define our comparator, which we refer to as the "more preferred" comparator, as follows. Let $s : M/{\sim} \to \mathbb{R}$ map a constraint to its strength. (Instead of $\mathbb{R}$, any totally ordered set could be used). For $c_1, c_2 \in M/{\sim}$, if $s(c_1) > s(c_2)$ then c_1 is stronger than c_2. Let seq_s map a set of constraints to an ordered sequence of the constraint strengths assigned to each constraint by s:

$$\begin{aligned} &seq_s : \\ &\{[m_1], [m_2], \ldots, [m_k]\} \mapsto (s([m_1]), s([m_2]), \ldots, s([m_k])), \\ &\qquad\qquad \text{where } s([m_1]) \geq s([m_2]) \geq \cdots \geq s([m_k]) \end{aligned}$$

With this definition, we can define a comparator for solution graphs (using $>^d$ to denote the lexicographical "greater than" operator between sequences):

Definition 3 ("More preferred" relation for solution graphs)**.** Let $G = \langle V + M, E\rangle$ be a constraint graph, $s : M/{\sim} \to \mathbb{R}$ a strength assignment function, and $G_1 = G[V + M_1]$ and $G_2 = G[V + M_2]$ two solution graphs of G. G_1 is *more preferred* than G_2 iff $seq_s(M_1/{\sim}) >^d seq_s(M_2/{\sim})$; we write $G_1 \succ_s G_2$.

Intuitively, we view the set of methods that are included in a solution graph as a sequence of strengths of the methods' constraints, ordered in decreasing order. These sequences are compared lexicographically: a greater sequence represents a stronger set of constraints. Note that the set of methods of a solution graph contains at most one method from each constraint, so, in the case of prop-

[2] This is not entirely accurate: a system with no constraints at all other than stay constraints for all of its variables, is satisfiable and has a unique plan.

erty models where the strenghts of stay constraints are unique, the strength of any retractable constraint appears at most once in a sequence. If, on the other hand, two or more retractable constraints have the same strength, it is possible that two different solution graphs map to the same sequence.

Using the above comparator, we can define the notion of a maximal solution graph of an over-constrained constraint graph with respect to a strength assignment function. We first note that $\succ_s$ is a partial order, in fact a strict weak order, so it makes sense to talk about a maximal element. This follows from defining $\succ_s$ via a homomorphism to the lexicographical comparison of ordered sequences of real numbers, itself a strict weak order (in fact a total order).

Definition 4 (Maximal solution graph)**.** Let $G = \langle V + M, E\rangle$ be a constraint graph, $s : M/\!\sim \to \mathbb{R}$ a strength assignment function, and $\mathcal{G}'$ the set of the admissible solution graphs of G. If G' is a maximal element in the strict weak order $(\mathcal{G}', \succ_s)$, we say that G' is a maximal solution graph of G.

1.5 Strength assignment function in property models

The constraint systems that arise in property models contain programmer specified constraints and stay constraints. The former kind are all assigned the same (strongest) strength *must*. Each stay constraint is assigned a weaker strength, unique among all constraints. This effects the "least surprising" behavior when interacting with a user interface.

1.6 Property model constraint system

Putting all the above together, we arrive at a complete formulation for expressing property models as constraint systems. In the definition below and later discussion, it is useful to be able to conveniently access the stay constraints of a constraint or solution graph $G = \langle V + M, E\rangle$. For this we define

$$stays(G) = \{\, [m] \mid m \in M, |outs(m)| = 1, |ins(m)| = 0\}.$$

Definition 5 (Property model constraint system)**.** If $G = \langle V + M, E\rangle$ is a constraint graph, $s : M/\!\sim \to \mathbb{R}$, $|stays(G)| = |V|$, $\forall c_i, c_j \in stays(G).c_i \neq c_j \implies s(c_i) \neq s(c_j)$, and $\forall c \in M/\!\sim \setminus stays(G).s(c) = must$, then we say that the pair $\langle G, s\rangle$ is a *property model constraint system*.

2. Solutions for Property Models

2.1 Solving algorithm

The algorithm we use for finding a maximal solution graph is a derivative of the *Quickplan* algorithm [7]. Quickplan is guaranteed to find a maximal solution graph if one exists, and fail otherwise. (Quickplan's "locally-predicate-better" predicate gives a different definition of maximal than the "more preferred" comparator we use; this difference does not manifest in property model constraint graphs.) For completeness, we describe the solver algorithm. Zanden's Quickplan is rather more complex than what we present here. Quickplan is "optimistic", starting from an over-constrained system, retracting (temporarily) constraints until a solution can be found. Then it improves the solution (if possible), by attempting to add retracted constraints back, one-by-one. We omit the first stage, and start directly from a system that we know can be satisfied, and go directly to the improvement phase. Further simplification results because each stay constraint has a distinct strength. The essence of the algorithm, nevertheless, remains the same.

In describing the algorithm, and later in the proof of the uniqueness of its result, we use the notions of *free* variable and method:

Definition 6 (Free variable, free method)**.** Let $G = \langle V + M, E\rangle$ be a constraint graph. A variable v is free in G if $\forall m_i, m_j \in M.m_i \in nbh(v) \wedge m_j \in nbh(v) \implies [m_i] = [m_j]$. A method m is free in G if $\forall v \in outs(m).v$ is free. We denote the set of free variables of a constraint $c \in M/\!\sim$ as $frees(c) = \{v \in V \mid nbh(v) = c\}$.

A free method can satisfy a constraint without restricting which method is used to satisfy any other constraints in the system.

We first describe the PLANNER algorithm (Algorithm 1) that finds a solution graph for a given constraint graph. This algorithm iterates adding a free method of the current constraint graph to the solution and removing the thus satisfied constraint from the current constraint graph—until all constraints are removed. The state of the algorithm is captured in two sets of methods: M_s are the methods that are part of the solution, and M_u the methods of the not yet satisfied constraints.

Algorithm 1 PLANNER($G\langle V + M, E\rangle$)

```
1: $M_s \leftarrow \emptyset$, $M_u \leftarrow M$
2: while $M_u \neq \emptyset$ do
3:     if no free methods in $G[V + M_u]$ then
4:         return "no solution"
5:     $m \leftarrow$ some free method in $G[V + M_u]$, $M_u \leftarrow M_u \setminus [m]$,
       $M_s \leftarrow M_s \cup \{m\}$
6: return $G[V + M_s]$
```

Algorithm 2, HIERARCHY SOLVER, finds a maximal solution graph given a constraint graph G and a strength assignment function s. We denote the maximal value of s, the strength of non-retractable constraints, as *must*. The HIERARCHY SOLVER algorithm first invokes PLANNER to solve one constraint system, in our case the original system without stay constraints. If this is not possible, the constraint system is not satisfiable. If a solution is found, the resulting plan is taken as the current best solution, which the algorithm tries to improve. On each iteration, the algorithm attempts to add one stay constraint c from the set of not yet processed constraints U to the constraint graph. If the graph stays satisfiable, the constraint is retained (added to the set of satisfied constraints S); if not, the constraint is retracted. When there are no more constraints to try, the solution graph of the most recent satisfiable constraint graph is the maximal solution graph.

Algorithm 2 HIERARCHY SOLVER($G\langle V + M, E\rangle, s$)

```
1: $S \leftarrow \{m \in M \mid s([m]) = must\}$, $U \leftarrow M \setminus S$
2: if PLANNER($G[V + S]$) has no solution then
3:     return "no solution"
4: while $U \neq \emptyset$ do
5:     $c \leftarrow \{m \in U \mid \forall m_i \in U.s([m_i]) \leq s([m])\}$
6:     if PLANNER($G[V + S \cup c]$) succeeds then
7:         $S \leftarrow S \cup c$
8:     $U \leftarrow U \setminus c$
9: return PLANNER($G[V + S]$)
```

With the assumptions that (1) the number of constraints that any variable belongs to, (2) the number of variables that any constraint involves, and (3) the number of distinct constraint strength values are all bound by some (small) constants, Zanden shows [7] that Quickplan's worst-case time complexity is $O(n^2)$, where n is the number of constraints in the system. In our case, condition (2) does not hold: the number of distinct constraint strengths is relative to the number of variables in the system. This assumption is not critical; even without it, $O(n^2)$ is an upper-bound of the asymptotic complexity of our solver algorithm because the loop in HIERARCHY SOLVER is linear in the number of stay constraints, and PLANNER is linear in the number of total constraints. Whether a solution exists at all can be determined in linear time.

Zanden proposes an "incremental" algorithm for solving a multi-way constraint system when a solution to a system that differs only in a small number of constraints is known [7]: two algorithms, one for finding a new solution graph when a new constraint is added to the constraint graph, one for when a constraint is removed, are provided. The worst-case complexity of these algorithms remains $O(n^2)$.

2.2 Uniqueness

Due to the special structure of property model constraint systems, if a maximal solution graph exists, it is unique—this is a crucial property for the predictability of user interfaces based on property models.

Zanden [7] showed the following property; the proof is short, so we include it for completeness.

Lemma 2.1 (Existence of a free method)**.** *A satisfiable constraint graph contains at least one free method.*

Proof. Let $G = \langle V + M, E\rangle$ be a constraint graph, and $G' = G[V + M']$ some solution graph of G. Let $E' = \mathit{edges}(G')$. To be acyclic, G' must have at least one method m such that $\forall v \in \mathit{outs}_{G'}(m).d^+_{G'}(v) = 0$. Assume no such method exists. Then for every method $m' \in M'$, $\exists v, m''.(m', v) \in E' \wedge (v, m'') \in E$. That is, from every method there is a path to some other, different, method. As there are a finite number of methods, the graph has a cycle; a contradiction.

A method that satisfies the above condition (outputs to only variables with no out-edges) is a free method. Assume m is such a method, and that $v \in \mathit{outs}(m)$, and v not free in G. Then $\exists m_1 \in M'.m_1 \in \mathit{nbh}_{G'}(v) \wedge m_1 \neq m$. The definition of solution graph requires that $d^-_{G'}(v) \leq 1$ and thus G' cannot contain both the edges (m, v) and (m_1, v). Therefore (v, m_1) must be included in G'. This contradicts that v has no out-edges, and thus m must be free.

The choice of the solution graph G' was arbitrary, thus the lemma follows. □

Lemma 2.2. *At most one solution graph $G' = \langle V + M', E'\rangle$ for the constraint graph $G = \langle V + M, E\rangle$ exists, such that $\forall v \in V.\exists m \in M'.(m, v) \in E'$.*

Proof. Lemma 2.1 guarantees that G' has at least one free method. We first show that for any free method m in G', $\mathit{outs}_G(m) = \mathit{frees}([m])$. Let $F = \mathit{frees}([m])$. Since m is free, $\mathit{outs}(m) \subseteq F$. Since $[m]$ is the only constraint attached to the variables F, and m is the only method in $[m]$ that exists in M', then for each variable v in the set $F \setminus \mathit{outs}(m)$, $\mathit{ins}_{G'}(v) = \emptyset$. However, by definition, for every variable v, $\mathit{ins}_{G'}(v) \neq \emptyset$. Therefore $\mathit{outs}_G(m)$ cannot be a proper subset of F, and must be equal to F. Because of the WF-4 condition in definition 1, $[m]$ does not contain other free methods in G.

We have now shown that every satisfiable constraint graph must contain at least one free method, which is the unique choice for a particular constraint in any solution graph. We can remove this constraint from G, and obtain a new, smaller, well-formed constraint graph $G_1 = G[V \setminus \mathit{frees}(m) + M \setminus [m]]$. The lemma is true for G iff the lemma is true for G_1. For an empty constraint graph (no variables, methods, or edges) the lemma is trivially true. □

Theorem 2.3 (Uniqueness of maximal solution graph)**.** *If $\langle G, s\rangle$ is a property model constraint system, then at most one maximal solution graph of G with respect to s exists. We call it the* most-preferred *solution graph of $\langle G, s\rangle$.*

Proof. Let $\langle G, s\rangle$ be a property model constraint system. Let $G_1 = \langle V + M_1, E_1\rangle$ and $G_2 = \langle V + M_2, E_2\rangle$ be maximal solution graphs of G with respect to s. We assume $G_1 \neq G_2$ and show a contradiction. We first show that G_1 and G_2 necessarily enforce the same constraints. Since all admissible solution graphs enforce all non-stay constraints, G_1 and G_2 could only differ in what stay constraints they enforce. Since the strength of each stay constraint is unique, $\mathit{stays}(G_1) \neq \mathit{stays}(G_2) \implies \mathit{seq}_s(\mathit{stays}(G_1)) \neq \mathit{seq}_s(\mathit{stays}(G_2))$, and thus either $G_1 \succ_s G_2$ or $G_2 \succ_s G_1$, violating the assumption that both solutions are maximal. Therefore, $\mathit{stays}(G_1) = \mathit{stays}(G_2)$.

It remains to show that $M_1 = M_2$, i.e., that if two maximal solution graphs enforce the same set of constraints, the methods selected from each constraint are the same. Every variable in a maximal solution graph is in the output set of some method. Consider the subgraph $G' = G[V + M']$ of $G = \langle V + M, E\rangle$ where $M' = M \setminus \{m \mid [m] \notin \mathit{stays}(G_1)\}$. That is, G' is G excluding the stay methods that do not appear in the solution G_1. The solution graphs of G' where $\forall v \in V.\exists m \in M'.(m, v) \in \mathit{edges}(G')$ (every variable is an output of some method) are the maximal solution graphs of G with respect to s. According to lemma 2.2, at most one such solution graph exists. □

2.3 Changes between consecutive solution graphs

The solution graph holds the potential set of functional dependencies, represented by edges, among variables in the constraint system at any one time. During execution of the methods, some of the dependencies could go unrealized, e.g., because a method did not use one of its inputs, but without statically analyzing the methods, the conservative assumption must be that every functional dependency in a solution graph is real. Therefore, by examining the potential changes in the solution graph for a constraint system, we are examining the potential changes in the values of variables in the constraint system.

2.3.1 When the solution graph does not change

Certain changes to a constraint system do not alter the most-preferred solution graph. Among other applications, this analysis can be used to avoid the work of computing a new solution graph. We focus on the effect of changes to the relative order of stay constraints in the constraint hierarchy, that is, changes to the strength assignment function. In particular, the following result is used for this purpose.

Lemma 2.4 (Increasing the strength of an enforced stay constraint does not change the most preferred solution graph)**.** *Let $\langle G, s\rangle$ be a property model constraint system where $G = \langle V + M, E\rangle$, G' its most-preferred solution graph, and c' some constraint in $\mathit{stays}(G)$ that is enforced in G'. Let s' be a strength assignment function such that $\forall c_1, c_2 \in M/\!\sim \setminus \{c'\}.s(c_1) < s(c_2) \implies s'(c_1) < s'(c_2)$, $\forall c \in \mathit{stays}(G) \setminus \{c'\}.s(c') > s(c) \implies s'(c') > s'(c)$, and $\langle G, s'\rangle$ a property model constraint system. The most-preferred solution graph of $\langle G, s'\rangle$ is G'.*

Proof. Let $G = \langle V + M, E\rangle$, $\langle G, s\rangle$ and $\langle G, s'\rangle$ property model constraint systems, G' the most preferred solution graph of $\langle G, s\rangle$, and c' a constraint in G'. The assumption $\forall c_1, c_2 \in M/\!\sim \setminus \{c'\}.s(c_1) < s(c_2) \implies s'(c_1) < s'(c_2)$, states that the relative order of strengths of two constraints, other than c', does not change from s to s'. Let us consider the case where the change from s to s' is such that c' "jumps" over exactly one constraint in the ordering induced by the strength assignment: $\exists! c \in M/\!\sim \setminus \{c'\}.s(c') < s(c) \wedge s'(c') > s'(c)$. Name the most-preferred solution graph of $\langle G, s'\rangle$ as G''.

If we show that G' and G'' enforce the same stay constraints, then the second half of the proof for Theorem 2.3 shows that

$G' = G''$. All constraints other than c and c' retain in G'' the enforcement status they held in G' because their relative order with all other constraints is unchanged. It remains to show that c', after becoming stronger in s', and c, after becoming weaker, are the same way. It should be obvious that a constraint (c') that is enforced when it is stronger than some set of constraints (call it C) should remain enforced after becoming stronger than a superset of those constraints ($C \cup \{c\}$). If c is retracted in G', it clearly cannot be enforced in G'' where it is weaker. If c is enforced in G' where c' did not need to be retracted, then c must be enforced in G'' as well.

Finally, the above change to s where c' moves over exactly one stronger constraint can be repeated arbitrarily many times, from which the lemma follows. □

The lemma implies that several other transformations can preserve the most preferred solution graph:

- a retracted stay constraint can be given a weaker strength;
- any contiguous region of retracted stay constraints can be permuted arbitrarily; and
- any contiguous region of enforced stay constraints can be permuted arbitrarily.

Finally, we observe that if G' and G'' are the most-preferred solution graphs of $\langle G, s' \rangle$ and $\langle G, s'' \rangle$, respectively, then $stays(G') \not\subset stays(G'')$ and $stays(G'') \not\subset stays(G')$. This observation may be useful in pruning the search space of all most-preferred solution graphs, e.g., when reasoning about all possible behaviors of a property model based user interface.

2.3.2 When the solution graph does change

It is possible to determine the extent of changes to the most-preferred solution graph induced by changes to the strength assignment function. In the special case of constraint systems for property models, the strength assignment function changes in a limited, predictable way: when a variable is edited (that is, its value is changed from outside of the constraint system), its stay constraint is given highest strength; all other strengths are left unchanged. Given this fact, the full set of changes possible to a solution graph after one edit can be found by, for each variable, (1) giving that variable's stay constraint highest strength, (2) finding a new most-preferred solution graph, and (3) identifying changes. Fortunately, there is a much simpler algorithm to perform this search.

Van Zanden observed that whenever QuickPlan attempts to enforce a previously retracted constraint, it must examine only the "upstream" constraints [7, §5.2]. A constraint c_1 is upstream from another constraint c_2 if, in the solution graph, there is a path from the selected method in c_1 to any of the output variables of any method in c_2. Since a stay constraint has only one method with one output, its upstream constraints are all the constraints with methods in its variable's ancestor graph. The *ancestor (directed acyclic) graph* of a variable v consists of every variable reachable from v in the transpose of the solution graph. Therefore, after a variable is edited, the only constraints that may need different satisfying methods are those constraints with methods in the variable's ancestor graph.

References

[1] A. Borning, R. Duisberg, B. Freeman-Benson, A. Kramer, and M. Woolf. Constraint hierarchies. *SIGPLAN Not.*, 22(12):48–60, 1987.

[2] B. N. Freeman-Benson, J. Maloney, and A. Borning. An incremental constraint solver. *Commun. ACM*, 33(1):54–63, 1990.

[3] J. Järvi, M. Marcus, S. Parent, J. Freeman, and J. N. Smith. Property models: from incidental algorithms to reusable components. In *GPCE '08: Proceedings of the 7th international conference on Generative programming and component engineering*, pages 89–98, New York, NY, USA, 2008. ACM.

[4] M. Sannella. Skyblue: A multi-way local propagation constraint solver for user interface construction. In *UIST '94: Proceedings of the 7th annual ACM symposium on User interface software and technology*, pages 137–146, New York, NY, USA, 1994. ACM.

[5] M. J. Sannella. *Constraint satisfaction and debugging for interactive user interfaces*. PhD thesis, University of Washington, Seattle, WA, USA, 1994.

[6] G. Trombettoni and B. Neveu. Computational complexity of multi-way, dataflow constraint problems. In *IJCAI (1)*, pages 358–365, 1997.

[7] B. V. Zanden. An incremental algorithm for satisfying hierarchies of multiway dataflow constraints. *ACM Trans. Program. Lang. Syst.*, 18(1):30–72, 1996.