Multi-Garnet: Integrating Multi-Way Constraints with Garnet

Michael Sannella and Alan Borning

Technical Report 92-07-01
Dept. of Computer Science and Engineering
University of Washington
September 1992

Abstract

Constraints provide a useful mechanism for maintaining relations in user interface toolkits. Garnet is a widely-used user interface toolkit with considerable functionality, based on one-way, required constraints. Multi-Garnet extends Garnet by adding support for multi-way constraints and constraint hierarchies with both required and preferential constraints. This document contains three chapters describing Multi-Garnet:

- Chapter 1 presents a high-level overview of Multi-Garnet. To motivate the development of Multi-Garnet, we examine the Garnet constraint system, present some realistic user interface problems that are difficult to handle in Garnet, and demonstrate how Multi-Garnet addresses these problems. We provide details on how Multi-Garnet supports some of the features of Garnet, including constraints with pointer variables, and inheritance of constraints.
- Chapter 2 contains a reference manual for the current version of Multi-Garnet (version 2.1). This includes information on compiling and loading Multi-Garnet, as well as documentation for the functions and macros used to create and manipulate Multi-Garnet constraints. This chapter also contains additional details on the implementation of Multi-Garnet.
- Chapter 3 describes a large Multi-Garnet example: a scatterplot displaying a set of points. Multi-Garnet constraints are used to maintain relationships between the data values, the screen positions of the points, and the positions of the X and Y-axes. Multiple interaction modes allow manipulating the scatterplot points and axes in different ways.

Contents

1 Multi-Garnet: Integrating Multi-Way Constraints with Garnet **1**

1.1 Introduction 1

1.2 Local Propagation Solvers 2

1.3 The Garnet Constraint System 3

1.3.1 Some Problems with Garnet Formulas 4

1.4 DeltaBlue and SkyBlue 5

1.5 Multi-Garnet 6

1.5.1 Multi-Garnet Constraints 6

1.5.2 Constraint Propagation 8

1.5.3 Indirect Reference Paths 8

1.5.4 Interactors 10

1.5.5 Coexistence of Garnet and Multi-Garnet 10

1.5.6 Inheritance 10

1.5.7 Performance 11

1.6 Future Work 12

1.6.1 Generalized Slot Accessers 12

1.6.2 Initializing Slots 12

1.6.3 Constraint Cycles 13

1.6.4 Debugging Tools 13

1.6.5 Applications 14

2 Multi-Garnet Version 2.1 Reference Manual **17**

2.1 Introduction 17

2.2 The SkyBlue Constraint Solver 18
2.3 Creating Constraints 18
2.4 Creating Stay Constraints 20
2.5 Storing and Using Constraints 20
2.6 Setting a Constrained Slot 21
2.7 Temporarily Setting and Anchoring Slots 22
2.8 Indirect Reference Paths 23
2.9 Inheriting Constraints, Indirect Path Slots, and Variable Slots 25
2.9.1 The Duplicate Constraint Problem 26
2.10 Interactors and Multi-Garnet Constraints 26
2.11 Coexistence of Garnet and Multi-Garnet 27
2.12 Errors under Multi-Garnet 28
2.13 Examining Constraints and Variables 29
2.14 Creating and Using Plans 29
2.15 Enabling and Disabling Multi-Garnet 31

3 A Scatterplot in Multi-Garnet 33

3.1 The Scatterplot Window 33
3.2 The "Use Plans" Button 35
3.3 The Scatterplot Constraint Network 35
3.4 The "Change Data" Interaction Mode 36
3.5 The "Move Points" Interaction Mode 37
3.6 The "Scale Points" Interaction Mode 38
3.7 The "Move Axis" Interaction Mode 38
3.8 The "Scale Axis" Interaction Mode 39
3.9 The "Move Axis & Points" Interaction Mode 40
3.10 The "Scale Axis & Points" Interaction Mode 41
3.11 The "X-Scale = Y-Scale" Button 42

Bibliography 47

Chapter 1

Multi-Garnet: Integrating Multi-Way Constraints with Garnet

1.1 Introduction

Constraints have become generally accepted as providing a useful mechanism for maintaining relations in user interface toolkits. Constraints can be used, for example, to maintain consistency between application data and a display of that data, to maintain consistency among multiple views of data, to maintain layout relationships among graphical objects, and to compose complex user interface components from simpler ones. Giving the system responsibility for maintaining the various relationships in a user interface frees the programmer from the tedious and error-prone task of maintaining these relationships by hand, making it easier to develop and maintain complex graphical user interfaces. User interfaces and user interface construction systems that use constraints include Peridot [Mye87], GROW [Bar86], Garnet [MGD+90a, MGD+90b], MEL [Hil90], RENDEZVOUS [Hil92], GITS [Ols90], Animus [Dui87], ThingLab II [Mal91], the Constraint Window System (CWS) [EL88], the FilterBrowser user interface construction tool [EMB87], and the Cactus statistics exploration environment [MSB90].

One significant difference among the constraint systems used in these user interface toolkits is whether they support one-way or multi-way constraints. For example, suppose we have a constraint that *window1* should be directly above *window2*. A one-way constraint would allow a new location for one of the windows (e.g. *window2*) to be determined given a change to the other window (*window1*), but not vice versa; a multi-way constraint would allow either *window1* or *window2* to be changed, and adjust the other window to re-satisfy the constraint. Similarly, a one-way constraint that $c = a + b$ would only allow a new value to be found for one of the variables (e.g. c) that satisfies the constraint, while a multi-way constraint could in general be used to find a new value for any of a, b, or c given values for the other two variables. Some one-way solvers allow cycles so that the $c = a + b$ constraint could be represented as three one-way constraints that can be used to find a value for a, b, and c respectively. In this case, multi-way constraints are preferable from the point

of view of software engineering, since what is conceptually a single relation can be represented as a single object in the system rather than three. Other one-way constraint satisfiers don't allow cycles in the constraint graph, in which case multi-way constraint systems are clearly more powerful and expressive than one-way ones.

Constraint hierarchies extend the basic theory of constraints to allow preferential constraints as well as required ones, with an arbitrary number of levels of preference [BMMW89]. In user interface applications, constraint hierarchies are useful for expressing declaratively the programmer's preferences about what is to be adjusted when resatisfying constraints after a change (since there are typically many ways of doing this), and for expressing defaults that can be overridden if necessary. An efficient incremental algorithm—DeltaBlue—is available for satisfying constraint hierarchies using local propagation [FBMB90].

Despite the advantages of multi-way constraints and constraint hierarchies, there is some feeling in the UI community that multi-way constraint solvers are significantly slower than one-way ones; or that even if they can be made efficient, they can only be embedded in systems with specialized and severely restricted architectures, thus rendering them unsuitable for general use. We attempt to dispel both of these impressions in this chapter.

Garnet is a widely-used user interface toolkit, built on Common Lisp and X windows, with considerable functionality [MGD+90a, MGD+90b]. However, Garnet supports only one-way constraints, all of which must be required (no hierarchies). In Multi-Garnet, we integrate both multi-way constraints and constraint hierarchies with Garnet. In the remainder of this chapter, we discuss the problems that arose in making this integration, and our solutions to these problems—information that should be of interest not only to the Garnet community, but also to other researchers who may wish to make use of multi-way constraints in user interface toolkits. In particular, Garnet supports pointer variables in constraints [VZMGS91], a facility that has proven useful for the dynamic runtime creation and manipulation of application objects. Our previous algorithms and systems haven't supported fully general pointer variables; Multi-Garnet does. In addition, Multi-Garnet uses a new constraint satisfaction algorithm, SkyBlue, which incorporates some important advances over DeltaBlue. We also discuss realistic user-interface problems that Garnet's constraint system cannot handle that are addressed by Multi-Garnet. We present information on the performance of Multi-Garnet, and show that it is competitive in performance with the existing system. We describe directions for future work, including the use of Multi-Garnet in a system for debugging constraint networks.

Multi-Garnet represents an advance over Garnet, since it includes multi-way constraints and constraint hierarchies. General support for pointer variables also makes it more powerful than our own previous systems such as ThingLab II. Because Multi-Garnet is built on top of Garnet, it has access to many other useful features not directly related to constraint technology, such as a library of pre-defined widgets and applications. This library will be used to create further Multi-Garnet applications, including debugging tools.

1.2 Local Propagation Solvers

A variety of techniques are available to solve collections of constraints. In user interface toolkits, by far the most common such technique is *local propagation*. (It is used in all of the systems cited in Section 1.1 except GITS.) For use with a local propagation solver, the object representing a constraint includes one or more *methods*, where each method takes the values of a subset of

the constrained variables (the method inputs) and calculates values for a disjoint subset of the constrained variables (the method outputs). If the output variables are set to these values, the constraint will be satisfied. For example, the constraint $a + b = c$ would in general include three methods: $c \leftarrow a + b$, $a \leftarrow c - b$, and $b \leftarrow c - a$. If the value of a or b were changed, the constraint solver could maintain the constraint by executing the first method to calculate a new value for c. If c were also constrained by $c + d = e$, then one of this other constraint's methods ($e \leftarrow c + d$) could be executed, and this process would continue until the change had propagated through the network of constrained variables.

Local propagation solvers cannot maintain all possible sets of constraints, for example, those involving simultaneous equations. However, local propagation solvers have the advantage that they are efficient and very general, since the code defining a method can do arbitrary computation. For example, in a user interface one might want to maintain the relationship between a string naming a font, and the object representing the font. The methods representing this constraint could call system functions for scanning font directories and reading in font definitions, in order to create a new font object that corresponds to a given font name string.

1.3 The Garnet Constraint System

The Garnet user interface development system is built on a prototype-based object system that supports one-way constraints between the slots (fields, instance variables) of objects. In addition to containing a value, each object slot may have an associated *formula* that calculates the slot value as a function of other slot values. When a slot value is changed, a local propagation solver can maintain the constraints by evaluating the code in the formulas.

```
(create-instance 'rect-proto opal:rectangle
   (:left 0) (:top 0) (:width 10) (:height 10)
   (:right (formula (+ (gvl :left)
                       (gvl :width)))))

(create-instance 'rect1 rect-proto
   (:left 50) (:top 10))

(create-instance 'rect2 rect-proto
   (:main rect1)
   (:left (formula (gvl :main :left)))
   (:top 30))
```

Figure 1.1: Garnet Objects and Formulas

Figure 1.1 shows the definitions of several rectangle objects with formulas associated with some of their slots. Formulas reference other slots in the object using the `gvl` form (`gvl` stands for "get value starting from local slot"). In the `rect-proto` object, the `:right` slot is calculated using the values of the `:left` and `:width` slots. The `gvl` form can also reference slots in other objects, by specifying a series of slots. In `rect2`, the `:left` slot value is calculated by accessing the `:main` slot which contains an object, and then accessing the `:left` slot of this object. In this case, the formula

causes `rect2` to be left-aligned with `rect1`. If the `:left` slot of `rect1` is changed, `rect2`'s `:left` formula will be recalculated using this new value. The `:main` slot can also be changed, to make `rect2` line up with a different object. This demonstrates the use of pointer variables in formulas.

Because the object system is prototype-based, any object can be used as a parent of further objects. The constraint system is fully integrated with the inheritance mechanism, allowing formulas to be inherited, in addition to slot values. In Figure 1.1, the formula associated with `rect-proto`'s `:right` slot is defined once in `rect-proto`, and inherited by the other rectangle objects. Inherited formulas reference local slots—the inherited formula for `:right` in `rect1` references the `:left` and `:width` slots of `rect1`, not `rect-proto`. In this case, the `:width` slot value of `rect-proto` is also inherited by `rect1`.

1.3.1 Some Problems with Garnet Formulas

Garnet formulas are a powerful tool for constructing user interfaces. However, there are some useful constraints that are hard to represent with formulas. For example, consider the definition of `rect-proto` in Figure 1.1. The `:right` slot formula allows calculating this slot from the `:left` and `:width` slots. Suppose that sometimes one wanted to position this rectangle by specifying the `:left` and `:right` slot values, or the `:width` and `:right` slots. Formulas could be added to the `:left` and `:width` slots to calculate each slot from the other two, and Garnet would correctly terminate the propagation cycles introduced. However, this may not always produce the expected results. If the `:left` slot value is changed, either the `:width` slot or the `:right` slot might be changed to maintain the constraint between these slots. It is possible to obtain either result, by setting and accessing slots in a certain order. However, to take advantage of this one has to understand the implementation of Garnet formulas. It would be much easier to maintain a user interface built with formulas if these choices could be specified as part of the formula definitions, so that they could be understood without reference to the internals of the system.

Another problem with using multiple formulas to represent multi-way constraints between slots is that the formulas to calculate each slot are defined separately, perhaps in different objects. As noted in Section 1.1, this represents a software engineering problem, placing the additional burden on the programmer of ensuring that all of the formulas representing a multi-way constraint are defined consistently, and are inherited together. It would be easier to define and use a multi-way constraint if all of the multiple formulas (methods) were bundled together into a single unit.

A more serious problem with Garnet formulas arises from the restriction that only one formula can be associated with a slot. In Figure 1.1, `rect2` is left-aligned with `rect1`. If `rect1`'s `:left` slot is changed, `rect2`'s `:left` slot will be changed accordingly. Suppose that one also wanted this to work in the other direction, so that changes to `rect2`'s `:left` slot would propagate to `rect1`'s `:left` slot. This could be done by adding a formula to `rect1`'s `:left` slot, producing a cycle between the two slots. Now, suppose that one wanted to add one or more new rectangles, left-aligned with both `rect1` and `rect2`, so that any of the object's `:left` slots could be changed while maintaining the alignment. This could be done by changing the existing formulas to construct a cycle between the `:left` slots of all of the objects, but it would require changing the existing network of formulas every time another object is added. It would be more convenient to add new left-aligned objects by simply adding formulas between each new object's `:left` slot and `rect1`'s `:left` slot. But this cannot be done with Garnet formulas, because `rect1`'s `:left` slot can only have one formula associated with it.

These problems suggest that Garnet could profitably use a more powerful constraint solver that allows defining the methods of a multi-way constraint in one place, and manages situations where multiple constraints can set a single slot.

1.4 DeltaBlue and SkyBlue

The DeltaBlue constraint solver addresses the problems with Garnet formulas mentioned above. In DeltaBlue, a constraint between a set of variables is represented by a set of methods, each of which calculates the value of an output variable using the values of the remaining variables so that the constraint is satisfied. More than one constraint may have methods that output to a given variable: it is the responsibility of the constraint solver to decide which methods to execute to satisfy the constraints.

DeltaBlue also solves networks containing both required and preferential constraints, allowing an arbitrary number of levels in the constraint hierarchy. All of the required constraints must be satisfied; if one cannot, DeltaBlue signals an error. At the non-required levels, a weak constraint is satisfied only if it doesn't conflict with a stronger one. Two or more constraints at the same level may also be in conflict; in this case, DeltaBlue arbitrarily picks one or the other to satisfy. (In the terminology of the constraint hierarchy theory given in [BMMW89, FBMB90], DeltaBlue produces a single *locally-predicate-better* solution to the hierarchy.)

Another important distinction between DeltaBlue and the Garnet constraint solver is that DeltaBlue supports separate planning and execution stages. During the planning stage, methods from the constraints are chosen and ordered; during the execution stage, given one or more starting input values, these methods are actually used to compute new values for the variables that will again find a locally-predicate-better solution to the constraints. In contrast, Garnet uses blind local propagation, with no separate planning stage. There are two advantages to DeltaBlue's approach. First, planning is necessary to find correct solutions if constraint hierarchies are permitted—otherwise blind propagation may choose the wrong constraint propagation path, resulting in some preferential constraints left unsatisfied that should be satisfied. Second, in a typical interactive graphics application, often a new input value is repeatedly fed into the same network of constraints—for example, when moving a part of a picture with the mouse. In such a case, the same plan can be executed many times, resulting in much better performance [Mal91].

We recently revised DeltaBlue to produce SkyBlue. Like its predecessor, SkyBlue finds a single locally-predicate-better solution to a constraint hierarchy, using local propagation and separate planning and execution stages. However, whereas DeltaBlue methods can only have one output variable, SkyBlue allows constraint methods to have multiple output variables. This is useful for expressing a number of important classes of constraints, in particular bidirectional constraints that can unpack a compound data structure into multiple variable values, or (running in the other direction) pack a suitable set of values into a compound object. SkyBlue also handles cycles of constraints in a more graceful fashion than DeltaBlue, which will make it possible to link in specialized, more powerful constraint solvers as needed (see Section 1.6.3). Additional details of SkyBlue are given in [San92].

1.5 Multi-Garnet

At the beginning of the project, we set a number of goals for Multi-Garnet:

1. To extend Garnet to use the SkyBlue constraint solver.
2. To change Garnet's programming style as little as possible, so that existing Garnet programs could be adapted without major changes. In particular, two subgoals were to continue supporting constraints with pointer variables, and to continue supporting inheritance of constraints in Garnet's prototype-based object system.
3. To allow the SkyBlue constraint solver to coexist with the Garnet formula constraint solver. For now, it is useful to be able to run existing Garnet programs without change, and build Multi-Garnet debugging tools using the Garnet library of pre-defined widgets. Eventually, it should be possible to replace all formulas with SkyBlue constraints, and remove the formula solver from Multi-Garnet.

These goals have been achieved. The following sections describe how constraints are represented and maintained in this system. These details are not just of interest to Multi-Garnet users. The strategies used to integrate multi-way hierarchical constraints into Garnet could be applied when extending other systems to handle these types of constraints, or to integrate a constraint solver into a system that doesn't support constraints at all.

The development of Multi-Garnet is continuing. As additional user interfaces are developed using this system, we will gain experience about which features are most useful, and the best syntax for accessing these features.

1.5.1 Multi-Garnet Constraints

Figure 1.2 shows the definitions of several rectangle objects that define Multi-Garnet constraints among their slots; they are analogous to the Garnet definitions given in Figure 1.1. The object `rect-proto` contains a multi-way constraint between its `:left`, `:right`, and `:width` slots, which is also inherited by `rect1` and `rect2`. The object `rect2` contains a multi-way constraint equating its `:left` slot value with `rect1`'s `:left` slot (referenced indirectly through its `:main` slot).

The `m-constraint` form defines a constraint by specifying its strength, the slots constrained, and one or more methods containing code to calculate output slot values in terms of the other slots. In this chapter, we use the strengths `:required`, `:strong`, `:medium`, and `:weak`. Each constrained slot is specified by a `gvl` form for accessing the slot, along with a variable name used to refer to this slot in the methods. Local slots within the object containing the constraint may be specified by simply giving the slot name, as with the `:left` slot of the `:left-cn` constraint in `rect2`.

The `m-stay-constraint` form defines a special type of constraint that specifies that its output variable should not change. In the example above, a `:medium` strength stay constraint anchors the `:width` slot of the rectangles, so that the constraint solver will change the `:left` or `:right` slots rather than the `:width` slot, if this can be done without violating stronger constraints.

This example addresses all of the problems with Garnet formulas raised in Section 1.3.1. The constraint in slot `:left-right-width-cn` allows the rectangle to be positioned by setting any two

```
(create-instance 'rect-proto opal:rectangle
   (:left 0) (:top 0) (:width 10) (:height 10) (:right 10)
   (:lrw-cn (m-constraint :required (left right width)
                          (setf left (- right width))
                          (setf right (+ left width))
                          (setf width (- right left))))
   (:width-stay (m-stay-constraint :medium width)))

(create-instance 'rect1 rect-proto
   (:left 50) (:top 10) (:right 60))

(create-instance 'rect2 rect-proto
   (:left 50) (:top 30) (:right 60)
   (:main rect1)
   (:left-cn (m-constraint :required ((main-left (gvl :main :left))
                                      left)
                           (setf left main-left)
                           (setf main-left left))))
```

Figure 1.2: Multi-Garnet Constraints

of the three slots `:left`, `:right`, and `:width`, calculating the third value from the other two. If the `:left` slot value is changed by itself, the stay constraint on the `:width` slot determines that the `:right` slot will be changed to maintain the constraint between these slots. By specifying all of the methods of the constraint in one place, it is easier to ensure that they are defined consistently. Finally, additional constraints can reference `rect1`'s `:left` slot, allowing an arbitrary number of left-aligned rectangles to be created without changing the existing constraint network. For example, a third rectangle could be defined as an instance of `rect2`, inheriting its `:left-cn` constraint and `:main` slot to keep it left-aligned with `rect1`. In this case, setting the `:left` slot of any of the three rectangles will cause the other two to be changed, to keep the three left-aligned.

There are some subtle differences between Multi-Garnet constraints and Garnet formulas. Formulas are "associated" with the output slot whose value is calculated by the formula. Multi-Garnet constraints are stored as the actual value of an object slot, referring to both the input and output slots of the constraint methods indirectly. Constraints can be added and removed from objects using the same operations used to access regular slot values. For example, `(s-value rect2 :left-cn nil)` will remove the constraint between `rect1`'s and `rect2`'s `:left` slots. In contrast, Garnet requires a special operation to remove a formula from a slot.

Multi-Garnet methods can reference both input and output slots using multiple-slot indirect reference paths. This is more symmetric than formulas, which can only reference input slots indirectly. This means that it is possible to create a constraint in one object constraining slots that are all in other objects. This gives the programmer more flexibility in choosing which object to use for storing a constraint.

One useful feature of Garnet formulas that is not supported by Multi-Garnet constraints is that formulas can specify which input slots they access on the fly, as the formula code is being executed. In contrast, Multi-Garnet requires that the constrained slots be specified separately from the method definitions. The reason for this restriction is because SkyBlue currently needs to know *which* slots

are being constrained before it adds the constraint to the network. In the future, it may possible to extend SkyBlue to lift this restriction for slots that are only accessed as inputs. In any case, we believe that the other benefits provided by the SkyBlue solver more than compensate for this restriction.

1.5.2 Constraint Propagation

Garnet evaluates formulas using lazy evaluation when a slot determined by a formula is read. In contrast, Multi-Garnet maintains multi-way constraints using eager evaluation, so constraint propagation may occur whenever a constraint is added or removed from the network. Propagation may also occur when a slot value is set directly. This case is handled specially, so that the constraints are correctly maintained.

In Garnet, the macro `s-value` is used to set a slot to a value, detecting any formulas that need to be reevaluated. Multi-Garnet modifies the meaning of `s-value` so that setting a constrained slot may cause values to propagate through the constraint network. When a constrained slot is set with `s-value`, the following occurs: (1) an *input constraint* is created that will set the slot to the specified value, (2) this constraint is added to the network, possibly setting the slot and propagating the value through the network if the input constraint is strong enough,[1] (3) this constraint is removed, possibly allowing the slot to be set by another constraint.

A few techniques are used to make this process more efficient. Input constraints are saved and reused, so that a new constraint isn't created every time a slot is set. Also, an input constraint is not even added to the constraint network if it can be determined that it is not strong enough to be satisfied.

An important result of this implementation is that `s-value` is not guaranteed to set the slot to the given value. If the slot value is determined by stronger constraints, then it will not be possible to satisfy the input constraint. Even if the input constraint is satisfied, other constraints may reset the slot value when the input constraint is removed. If one wants to ensure that a slot is set to a given value, and is not changed, it is necessary to create a required constraint that sets the slot to the value. Multi-Garnet supplies macros that can be used to temporarily install constraints that set a slot to a given value, during the execution of a form.

1.5.3 Indirect Reference Paths

Garnet formulas can reference input slots by specifying an indirect reference path such as `(gvl :main :left)`. As noted above, continuing to support pointer variables was one of our design goals for Multi-Garnet. However, at first glance this seems incompatible with the DeltaBlue and SkyBlue algorithms, which assume that a constraint always constrains the same variables. Finding a solution to this problem is important not just for Multi-Garnet, but for any system that includes both multi-way constraints and pointer variables. Some of our earlier systems, in particular the original ThingLab [Bor81], made extensive use of paths for specifying which slots were affected by a constraint; in addition these paths interacted correctly with inheritance. However, they were always

[1] The input constraints used by `s-value` have a default strength (normally `:strong`). A slot can be set using an input constraint of any strength by calling the function `s-value-strength`, which takes an additional strength parameter.

relative to the part-whole hierarchy of the object that owned the constraint. Garnet pointers can specify arbitrary references, not necessarily just to a subpart, and can also be changed dynamically.

Multi-Garnet implements constraints with pointer variables by transforming them into constraints with fixed references to the final slots specified by the indirect reference paths. Such fixed-reference constraints can be added and removed from the network by calling SkyBlue. If one of the slots along an indirect reference path is changed, the associated fixed-reference constraint is removed from the network, this constraint is changed to refer to the new slots specified by the indirect reference paths, and the constraint is added to the network to constrain these new slots.

This approach raises some interesting issues. What should happen if indirect reference paths from more than one constraint use a particular slot that is changed? The order in which these constraints are removed and added to the network can affect the propagation path chosen and exact constraint solution found. Also, what should happen if an indirect reference path is broken, i.e. the path cannot be followed because one of the slots along the path does not contain a legitimate Garnet object? Garnet can simply ignore a formula with a broken path, but removing a constraint with a broken path from a constraint hierarchy may cause other constraints to be satisfied, and values to propagate. Finally, what should happen if a slot along an indirect reference path is constrained?

Multi-Garnet addresses these issues by distinguishing between *active* and *inactive* constraints. When a constraint is created and stored in a slot, its indirect reference paths are followed to determine which slots are being constrained. If none of the paths is broken, then the constraint is activated, and added to the constraint network. If a slot accessed along an indirect reference path changes, (1) all active constraints with an indirect reference path using that slot are removed from the constraint network and deactivated, (2) the indirect reference paths for all inactive constraints whose paths access that slot are re-evaluated, and (3) any of these constraints whose paths are now unbroken are activated and added to the constraint network. This scheme will automatically remove constraints when their indirect reference paths are broken, and add them again whenever the paths are re-connected.

The order in which the inactive constraint paths are evaluated and the constraints are added is deliberately left unspecified. If the order matters, this indicates that there are multiple possible ways to solve the constraint network, and the user may want to alter the constraint network to select a single desired solution.

Slots along an indirect reference path can be constrained by other constraints. When constraint propagation causes such a slot to be changed, constraint propagation is completed using the existing constraint network, then any constraints whose indirect reference paths have changed are updated as described above. This may cause further constraint propagation, and indirect reference path invalidation, etc. Note that it is possible that a constraint that will be removed (because a slot on its indirect reference path has changed) can still influence the constraint graph, and can still be used to propagate variable values, up until the moment it is removed.

This implementation of indirect reference paths may seem suspiciously simple, compared to the elaborate algorithms described in [VZMGS91]. There are two reasons for this simplicity. First, all constraint propagation is handled by the SkyBlue algorithm as constraints are added and removed from the network, so propagation wasn't explicitly described above. Second, in this scheme constraint value propagation and indirect pointer updates are completely separate—if a slot along an indirect reference path is changed during constraint propagation, the propagation is completed using the current constraint graph before any indirect pointers are updated. This is distinctly different from the Garnet constraint solver and the algorithms described in [VZMGS91], which allow the edges

in the constraint graph to be changed during constraint evaluation (which accounts for some of the complexity of these algorithms). Eventually, it may be possible to modify SkyBlue to directly support indirect pointers, interleaving constraint propagation and indirect pointer updates.

1.5.4 Interactors

User interaction is implemented in Garnet using *interactors* [Mye90] that interpret input events, such as key presses and mouse movements, and set specified slots in graphic objects, such as the position of an icon being moved by the mouse. These slots are set using `s-value`, so constraint propagation will occur if these slots are constrained by Multi-Garnet constraints. In most situations, normal Garnet interactors can be used in Multi-Garnet without change.[2] Some interactors may behave differently if the slots they set are constrained. For example, if an interactor tries to move a object being dragged with the mouse by setting its `:box` slot with `s-value` (which uses a default `:strong` input constraint), and this slot is constrained by a `:required` stay constraint, the object will not move.

1.5.5 Coexistence of Garnet and Multi-Garnet

Multi-Garnet was implemented by slightly modifying some of Garnet's internal functions to detect and handle Multi-Garnet constraints, while taking care not to remove any of the formula handling code. Regular Garnet applications that use formulas can be loaded and run in Multi-Garnet, coexisting with applications built entirely using Multi-Garnet constraints.

Garnet formulas and Multi-Garnet constraints can also be mixed in the same application, although all possible combinations of formulas and constraints are not supported. It is possible to create Garnet formulas that read the values of slots constrained by Multi-Garnet constraints. However, it is not always possible for Multi-Garnet constraints to constrain slots that are set by Garnet formulas. The latter functionality has been implemented on an incomplete and experimental basis to allow Multi-Garnet applications to use the existing library of Garnet gadgets built using formulas, without rewriting all of the formulas as constraints.

1.5.6 Inheritance

Multi-Garnet implements inheritance of multi-way constraints and constrained slots slightly differently than Garnet handles inheritance of formulas. First, constraints are only inherited when an object is created, by copying any constraints in the parent object slots down to the child. After a child is created, adding or removing constraints from the parent has no effect on the child. Second, any inherited slots accessed while following indirect reference paths (including the final slot to be constrained) are copied down when they are first accessed.

Eventually, it may be possible to modify Multi-Garnet to implement more sophisticated inheritance behavior, such as Garnet's ability to modify inherited formulas when the parent formula is changed. However, this raises some issues about the meaning of multi-way constraints in an object-oriented

[2] A few interactors needed to be slightly modified for use with Multi-Garnet because they destructively update data structures stored in a slot, rather than setting the slot using `s-value`. Garnet also had to take special steps to make formulas work properly in these situations.

system. For example, if a child object inherits a slot value, should it be able to constrain this slot without copying it down? If so, then this would violate the idea that the status of child objects should not affect the parent.

1.5.7 Performance

Performance is an issue for any system used as a base for interactive graphics applications. Our experience has been that applications built using Multi-Garnet constraints are just as fast as similar ones built using Garnet formulas. To test this impression, we constructed instrumented Garnet and Multi-Garnet versions of the `demo-manyobjs` program. This program, one of the standard test programs in the Garnet library, repeatedly moves a box connected by lines to other boxes, thus causing Garnet formulas (or Multi-Garnet constraints) to be re-evaluated repeatedly to determine the endpoints of the connecting lines. In addition to comparing the performance of Garnet and Multi-Garnet running `demo-manyobjs`, we measured the performance of Multi-Garnet when it was modified to use the DeltaBlue algorithm instead of SkyBlue. We also compared the performance of Multi-Garnet when it created and reused a single plan, versus when it calculated the constraint propagation paths repeatedly. The results, shown in Figure 1.3, give the time to move a box and update the screen 500 times.

These numbers were measured using Multi-Garnet (version 1.5), Garnet (version 1.4) and Allegro Common Lisp (version 4.0.1), running on a SUN SPARCstation IPX.

version	*time (seconds)*
Garnet	8.84
Multi-Garnet & DeltaBlue (no plan)	8.17
Multi-Garnet & DeltaBlue (plan)	7.54
Multi-Garnet & SkyBlue (no plan)	15.31
Multi-Garnet & SkyBlue (plan)	7.99

Figure 1.3: Garnet and Multi-Garnet Timings for `demo-manyobjs`

For this benchmark, the modified version of Multi-Garnet using DeltaBlue was faster than the Garnet version. Reusing plans caused even greater speed-up. Multi-Garnet with SkyBlue is not currently as fast as Multi-Garnet with DeltaBlue. However, the SkyBlue algorithm has not been optimized for speed; once this is done, we expect that it will be about as fast as DeltaBlue. Even with the unoptimized SkyBlue, however, when reusing saved plans the result is faster than standard Garnet.

The `demo-manyobjs` benchmark is a good test of simple constraint propagation. However it does not test all of the facilities of Multi-Garnet, since it only uses one-way required constraints, and does not dynamically change indirect reference pointers. It is difficult to directly compare Multi-Garnet programs using multi-way constraints and multiple strength levels to Garnet, since the Garnet program would have to be written differently to provide the same functionality (perhaps explicitly adding and removing constraints to correspond to different Multi-Garnet solutions). The technique Multi-Garnet uses to implement indirect references (Section 1.5.3) is probably slower than that of Garnet, but even in this case it is difficult to make a direct comparison, since Multi-Garnet also supports indirect reference paths on method outputs. Meaningful comparisons of Garnet and Multi-Garnet performance will have to measure the performance of real user interfaces.

One situation in which Multi-Garnet (or any other system using SkyBlue) could encounter performance problems is when the constraint network contains numerous constraints with multi-output methods. It was proved in [Mal91] that the problem of finding a locally-predicate-better solution to a constraint hierarchy is NP-complete when methods have multiple outputs. However, we hypothesize (and will be attempting to prove) that SkyBlue is in the worst case exponential-time only in the number of constraints with multi-output methods, not in the total number of constraints. Further, in the applications we have written so far in Multi-Garnet, in practice there is very little backtracking during SkyBlue's planning stage (the algorithm includes a potential backtracking step associated only with multi-output constraints). This suggests that even for multi-output constraints, this exponential-time behavior arises only for pathological cases. In any event, performance has not been a problem so far.

1.6 Future Work

Multi-Garnet is being developed as part of the first author's Ph.D. dissertation research, and work on Multi-Garnet itself, on underlying constraint satisfaction algorithms, and on tools and applications that use Multi-Garnet, is continuing.

1.6.1 Generalized Slot Accessers

Accessing a constrained slot by following an indirect reference path is a special case of the more general facility of accessing slots by evaluating an arbitrary form. Garnet formulas can contain conditionals and loops to calculate which input slots are accessed. For example, the formula that calculates the bounding box of an aggregate can loop through the list of children objects, accessing an arbitrary number of child bounding boxes. Based on experience with this feature in Garnet, we plan to extend Multi-Garnet to allow the constrained slot specifications to include arbitrary expressions. As these expressions are executed, they would access slots using a special form (such as `gvl`) that records the slots used. If a constrained slot cannot be accessed, the reference would be marked as "broken," and the constraint deactivated. This same mechanism could be used to implement *activation conditions,* forms associated with a constraint whose value determines whether a constraint should be active or not.

1.6.2 Initializing Slots

When creating an object with constraints between its slots, one would like to specify initial values for some of the slots, and use the constraints to set the other slots to corresponding values. For example, when creating a rectangle with a constraint between its `:left`, `:right`, and `:width` slots, one might want to specify two of these three slots, and have the other one computed.

This issue is currently not addressed in Multi-Garnet. When creating objects with constraints between their slots, it is best to initialize all of the slots to corresponding values, to prevent the solver from using the values of uninitialized slots. To address this problem, we plan to extend the `create-instance` operation to create temporary stay constraints on slots with initial values, to hold them constant while other constraints are added that reference these slots. This constraint

system initialization problem is not unique to Multi-Garnet, but has been noted by a number of other researchers (see e.g. [Mal91]); a solution to it would be of general utility.

1.6.3 Constraint Cycles

Local propagation works well in many situations, but runs into problems when the constraint network contains a cycle. Some local propagation solvers (including Garnet's formula solver) handle cycles by executing the methods once around the cycle, or repeatedly executing the methods around the cycle until the variable values stabilize. Neither of these techniques can guarantee that all of the constraints will be correctly maintained.

When the SkyBlue solver detects a cycle of methods, it prints a warning and marks as invalid all variables in the cycle (and all variables derived from these variables). If the cycle is later broken, the invalid variables are recalculated and marked as valid. We plan to extend SkyBlue to examine the constraints in a cycle, select a suitable constraint solver for these troublesome constraints, and pass the cycle off to that solver. For example, if all of the constraints in a cycle represent linear equations, then the constraints could be passed off to a linear equation solver that uses Gaussian elimination.

1.6.4 Debugging Tools

Multi-Garnet has been developed as part of research on debugging tools for hierarchical constraint networks. Constraint networks can be difficult to understand. It may not be obvious to the programmer how the constraints are being maintained, and how changed values propagate through the network. If a constraint network is not behaving as expected, it can be difficult to isolate which particular part of the network is causing the problem. We believe that this situation can be improved by creating debugging tools to manipulate and display the constraint network.

Multi-way hierarchical constraints are a particularly good domain for developing debugging tools, because the possible constraint propagation paths are specified precisely and declaratively by the constraint hierarchy. Debugging tools can use this information to analyze the constraint network and provide information to pinpoint possible trouble spots. For example, a debugger for Multi-Garnet could examine a constrained slot, and determine whether `s-value` could successfully change its value. In some simpler constraint systems, explaining the behavior of a constraint network requires exposing more details of the implementation of the constraint solver, since the specification of the solutions chosen is only implicit in the solver's code, and not given by a separate, declarative theory.

Debugging tools can provide many different types of information to a programmer trying to understand or debug a constraint network. Network display tools can be used to examine the state of the constraint network—prototype display tools were used to create Figure 1.4, which displays the constraint network used to relate the axes of a scatterplot to a single point. History tools can maintain information on how the slot values and the constraint network changes while an interaction is in progress, and provide facilities for tracing and undoing these changes. Explanation tools can analyze the network, to answer questions such as why a particular constraint is satisfied or unsatisfied. Using this information, network editing tools can allow the programmer to change the constraint network interactively, to fix bugs in a running system, and to experiment with possible changes to the constraint network.

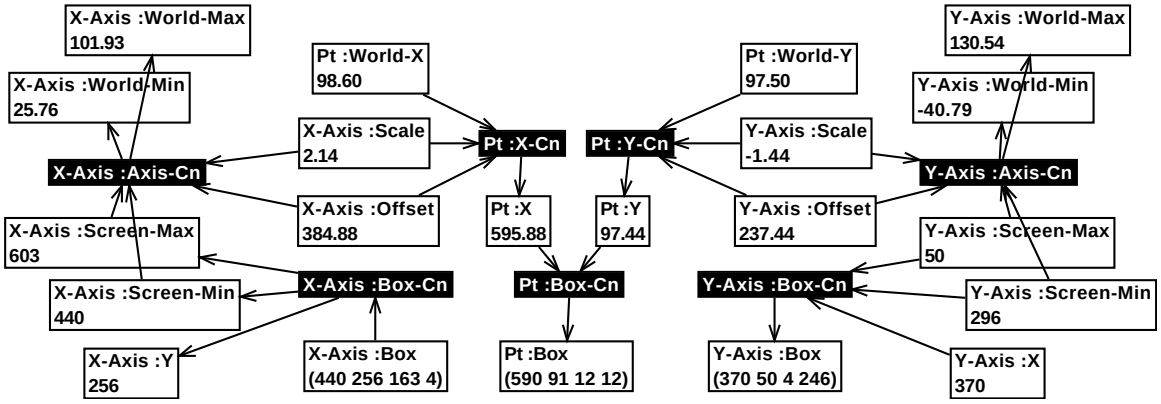
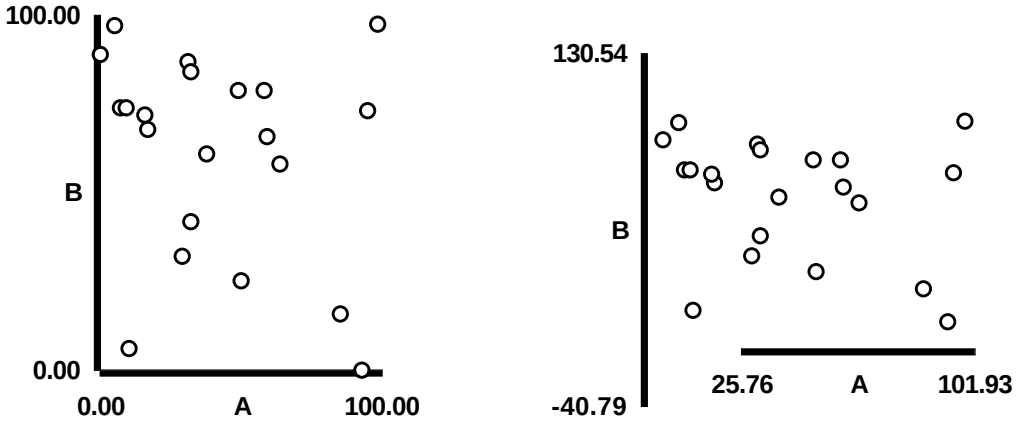


Figure 1.4: Scatterplot X-Axis and Y-Axis Networks

1.6.5 Applications

Multi-Garnet provides a good base for building complex user interfaces. One area where we plan to apply this system is to construct interactive data analysis systems [MSB90]. For example, Figure 1.5a shows a scatterplot displaying a collection of object measurements. Multi-Garnet constraints are used to maintain relations between the object measurements, the positions of points in the scatterplot, and the positions and ranges of the scatterplot axes. Figure 1.5b shows the same scatterplot, after the scatterplot points and axes have been moved and resized. This is still a meaningful scatterplot, displaying the same data. Figure 1.4 displays the constraint network used to relate the x and y-axes of this scatterplot to a single point.



(a) The Original Scatterplot (b) After Moving Points and Axes

Figure 1.5: A Scatterplot Built Using Multi-Garnet Constraints

Another planned application is the Electronic Encyclopedia Exploratorium (E^3), a new, multi-investigator project in which we are building "how things work" articles for an electronic encyclopedia [ABB+92]. Multi-Garnet will be used in constructing E^3's user interface. Typical articles in E^3 will describe such mechanisms as refrigerators, engines, telescopes, and mechanical linkages. Each article

will provide simulations, 3-dimensional animated graphics that the user can manipulate, laboratory areas that allow a user to modify the device or experiment with related artifacts, and a facility for asking questions and receiving customized, computer-generated English language explanations.

Acknowledgements

Thanks to Bjorn Freeman-Benson, Brad Myers, and Brad Vander Zanden for much helpful advice on this project and for comments on drafts of this chapter. This work was supported in part by the National Science Foundation under Grant Nos. IRI-9102938 and CCR-9107395.

Chapter 2

Multi-Garnet Version 2.1 Reference Manual

2.1 Introduction

The Multi-Garnet package integrates hierarchical multi-way constraints into Garnet. It is based on the SkyBlue algorithm, which supports multi-output constraints, and includes some support for cyclic constraints.

Multi-Garnet v2.1 has been developed and tested under Garnet version 2.0 and Franz Allegro Common Lisp version 4.1. It may require small changes to run under other versions of Lisp. It may require significant changes to run with newer versions of Garnet. As of this writing, it appears that Multi-Garnet v2.1 works correctly under the alpha release of Garnet v2.1. It will probably run under the final release of Garnet v2.1. Beyond that, we can't make any guarantees. We hope to update Multi-Garnet to work with future releases of Garnet.

The Multi-Garnet v2.1 release consists of the following files:

`load-multi-garnet.lisp`	Loads Multi-Garnet.
`compile-multi-garnet.lisp`	Compiles and loads Multi-Garnet.
`object-rep.lisp`	Object definitions.
`sky-blue.lisp`	SkyBlue constraint solver.
`multi-garnet.lisp`	Modifications to Garnet.
`examples.lisp`	Multi-Garnet examples.
`scatterplot.lisp`	Extended example.

The Multi-Garnet system code is defined in the files `object-rep.lisp`, `sky-blue.lisp`, and `multi-garnet.lisp`. To compile and load these files, load the file `compile-multi-garnet.lisp`. Once these files have been compiled, loading the file `load-multi-garnet.lisp` will load the Multi-Garnet system. Both of these files check that Garnet v2.0 is already loaded, and signal an error if it is not.

The files `examples.lisp` and `scatterplot.lisp` contain example code that uses Multi-Garnet. `examples.lisp` contains many small examples, that can be individually executed. To try them

out, load `examples.lisp` (which will load a few auxiliary functions), and then select and execute examples from the text of the file. `scatterplot.lisp` is described in detail in Chapter 3 of this document ("A Scatterplot in Multi-Garnet").

All of the Multi-Garnet system code is loaded into the package `:multi-garnet` (nicknamed `:mg`). All of the functions, macros, and variables mentioned in this document are exported from this package.

`*multi-garnet-version*` [*Variable*]

Value is a string identifying the currently-loaded version of Multi-Garnet. For Multi-Garnet v2.1, this is the string `"2.1"`.

2.2 The SkyBlue Constraint Solver

This version of Multi-Garnet is based on the SkyBlue constraint solver. The most significant differences in functionality between SkyBlue and the previously-used constraint solver (DeltaBlue) is that SkyBlue is more tolerant of cycles (see Section 2.12), and SkyBlue supports multiple-output methods. A few notes:

- The SkyBlue algorithm code is still somewhat rough (i.e. read it at your own risk). Soon we hope to restructure the code, and publish it in a technical report, along with more explanatory material and a proof of correctness.
- The SkyBlue code has not been optimized for speed yet. Therefore, any benchmark measurements made using this code would be particularly suspect.
- If there are constraints with multiple-output methods in the constraint network, adding and removing constraints may cause backtracking. It has been proved that solving such networks may take exponential time in the worse case, but usually this is not a problem.

`*sky-blue-backtracking-warning*` [*Variable*]

If this variable is non-`NIL`, a warning message will be printed whenever SkyBlue backtracking occurs. Initially `NIL`.

2.3 Creating Constraints

Constraints are created using the following macro:

`(m-constraint` *strength paths* `&rest` *methods*`)` [*Macro*]

The strength of the constraint is specified by *strength*, which should be one of the keywords `:required`, `:strong`, `:medium`, or `:weak`. Multi-Garnet currently only supports this fixed hierarchy of strengths. Future versions of Multi-Garnet may support changing the strength hierarchy.

paths is a list of indirect reference path specifications, each of the form `(`*var-name* `(gvl .` *slots*`))`, specifying the series of slots to follow to access one of the constraint variables. Indirect reference paths are interpreted by starting at the "root" object containing the constraint, and accessing each

slot in turn (see Section 2.8). If an indirect reference path is just specified by a symbol, this is equivalent to (*symbol* (gvl *keyword-symbol*)), i.e. xyz is the same as (xyz (gvl :xyz)).

methods is a list of forms, each defining a single method for the constraint. Single-output methods are specified by method definitions of the form (setf *var-name* . *forms*), where *var-name* is one of the variable names defined in *paths*, and *forms* is a series of arbitrary lisp forms that can access any of the *paths* variable names as free variables. When the method is run, the forms are evaluated, and the value of the last form is stored in the specified variable.

Multiple-output methods are specified by method definitions of the form (setf *var-names* . *forms*), where *var-names* is a list of variable names defined in *paths*, and *forms* is a series of arbitrary lisp forms that can access any of the *paths* variable names as free variables. When the method is run, the forms are evaluated, and the multiple values returned by the last form are stored in the specified variables.

Note: The symbols "gvl" and "setf" are just used to make the form easier to read. The macro kr:gvl is not executed to interpret the indirect reference paths.

Some examples:

```
(m-constraint :required (a b) (setf a b) (setf b a))
```

The above required constraint equates the value of the :a and :b slots of the constraint's root object.

```
(m-constraint :strong ((oleft (gvl :obj :left))
                       left delta)
              (setf left (+ oleft delta))
              (setf oleft (- left delta))
              (setf delta (- left oleft)))
```

The above strong constraint maintains a relationship between the slots :left and :delta of the constraint's root object, and the :left slot of whatever object is stored in the :obj slot of the root object.

```
(m-constraint :required (left top width height box)
              (setf box (list left top width height))
              (setf (left top width height)
                    (values (first box)
                            (second box)
                            (third box)
                            (fourth box))))
```

The above required constraint specifies that the :box slot of the the constraint's root object should contain a list of the values of the :left, :top, :width, and :height slots. If any of these four slots is changed, the first method changes the :box slot. If the :box slot value is changed, the second method (a multiple-output method) "unpacks" the :box list and sets the four other slots.

2.4 Creating Stay Constraints

A stay constraint is used to "anchor" the value of a variable, so it will not be changed unless the stay constraint is overridden by a stronger constraint. Stay constraints can be created using an `m-constraint` form such as `(m-constraint :required (a) (setf a a))`, which has the effect of creating a required stay constraint on the `:a` slot of the constraint's root object. However, it is better to define stay constraints explicitly using the `m-stay-constraint` macro:

`(m-stay-constraint` *strength* `&rest` *paths*`)` [*Macro*]

strength is the strength keyword, just as in `m-constraint`. *paths* is a list of indirect reference path specifications, either simple symbols (specifying a local slot name of the constraint's root object), or "`gvl`" forms (specifying a slot to be accessed indirectly).

Examples:

```
(m-stay-constraint :required a)
```

This creates a required stay constraint on the `:a` slot of the constraint's root object.

```
(m-stay-constraint :strong (gvl :a :b) (gvl :a :c))
```

This creates a strong stay constraint on the `:b` and `:c` slots of the object stored in the `:a` slot of the constraint's root object. Note that a stay constraint may specify multiple slots whose values should not be changed. In this case, the stay constraint will only be satisfied if *all* of the variables can be "anchored" simultaneously. If another stronger constraint overrides one of the variables anchored by a multiple-output stay constraint, the other variables will not be constrained by the stay constraint.

2.5 Storing and Using Constraints

To use a constraint, it must be stored in a slot of a KR object. This object provides the "root object" used to interpret the indirect reference paths used to access the constraint's variables. It is not necessary to explicitly activate a constraint. When a slot is set to a constraint by `s-value` or `create-instance`, the constraint is activated (unless one of its indirect reference paths is "broken," see Section 2.8). To remove a constraint, simply store something else in its slot, such as `NIL`.

At any given time, a particular constraint object can only "belong" to one root object and slot. For example, suppose that a constraint has been created and stored in the `:cn` slot of the object `foo`. At this point, the root object of this constraint is `foo`, and its home slot is `:cn`. If a pointer to the constraint is copied to the `:cn` slot of `bar` with `(s-value bar :cn (g-value foo :cn))`, the constraint will *not* be activated with root object `bar`. The association between the constraint and its root object can only be removed by explicitly storing something else in the `:cn` slot of `foo`, such as `(s-value foo :cn nil)`. The correct way to copy a constraint from `foo` to `bar` is to execute `(s-value bar :cn (clone-constraint (g-value foo :cn)))`. The function `clone-constraint` takes a constraint, and returns a new copy without a root object or home slot, that can be set into a slot to be activated.

Most of the time, constraints are simply created and stored in a single object slot, so keeping track of the association between the constraint and its root object is not an issue. However, programs that explicitly manipulate constraints need to take account of this behavior. The current root object and slot of a constraint can be determined by calling `constraint-state` (see Section 2.13).

Note that constraints are stored as the actual value of an object slot, as compared to formulas which are "associated" with a slot. This means that the slot containing a constraint cannot be used as one of the constraint's variables. Unlike formulas, a multi-way constraint does not have a distinguished output variable whose value is stored in the constraint's slot.

The reason for storing constraints in particular object slots is because the slots provide names for the constraints that can be used to access the constraints, remove them, and override inherited constraints. This is simpler than Garnet, which requires a special operation to remove a formula from a slot.

Examples:

```
(create-instance 'foo nil
                 (:a 5)
                 (:b 6)
                 (:eq-cn (m-constraint :required (a b)
                                       (setf a b)
                                       (setf b a)))
                 )
```

The above form creates an object `foo`, which includes a required constraint in its `:eq-cn` slot that ensures that its `:a` and `:b` slot contain the same value. If one of the slots is changed using `s-value`, then the change is propagated to the other slot.

```
(s-value foo :eq-cn nil)
```

This removes the above constraint, so `:a` and `:b` can be changed independently.

```
(s-value foo :stay-cn (m-stay-constraint :required a))
```

This adds a required stay constraint that anchors slot `:a`.

2.6 Setting a Constrained Slot

Garnet evaluates formulas using lazy evaluation when a slot determined by a formula is read. In contrast, Multi-Garnet maintains multi-way constraints using eager evaluation, so constraint propagation may occur whenever a constraint is added or removed from the network. Propagation may also occur when a slot value is set directly. This case is handled specially, so that the constraints are correctly maintained.

Garnet allows `s-value` to set the cached value of a slot with an attached formula, and uses an unintuitive set of rules to determine when the cached value is reset. With multi-way constraints, setting a slot that has an attached constraint may cause constraint solving and propagation.

In Multi-Garnet, when a constrained slot is set with `s-value`, the following occurs: (1) an "input constraint" is created (with strength `*default-input-strength*`, normally `:strong`) that will set the slot with the specified value, (2) this constraint is added to the network, possibly setting the slot and propagating the value through the network if the constraint's strength is strong enough, (3) this constraint is removed, possibly causing the slot to be set by another constraint.

A few techniques are used to make this process more efficient. Input constraints are saved and reused, so that a new constraint isn't created every time a slot is set. Also, if Multi-Garnet can determine that the input constraint will not be satisfied (because its strength is too low), then it is not even added to the network.

An important result of this implementation is that `s-value` is not guaranteed to set the slot to the given value. If the slot value is determined by stronger constraints, then it will not be possible to satisfy the input constraint. Even if the input constraint is satisfied, other constraints may reset the slot value when the input constraint is removed.

(`s-value-strength` *obj slot value strength*) [*Function*]

The function `s-value-strength` sets an object slot the same as `s-value`, except that the specified strength is used when attempting to set the value of the slot, instead of `*default-input-strength*`.

2.7 Temporarily Setting and Anchoring Slots

Often, it can be useful to temporarily anchor slots, or set them to a particular value, during the execution of some forms. This could be done by explicitly creating constraints, storing them in object slots, and then removing them, but the following macros make this easier:

(`with-stays` *stay-specs* `&rest` *forms*) [*Macro*]

stay-specs is a list of lists of the form (*obj slot strength*), specifying the object slots that are to have stay constraints with the specified strength applied to them. If *strength* is not specified, it defaults to the value of `*default-input-strength*`, initially `:strong`.

(`with-slots-set` *slot-set-specs* `&rest` *forms*) [*Macro*]

slot-set-specs is a list of lists of the form (*obj slot value strength*), specifying the object slots that are to be constrained, the value that the slot should be constrained to have, and the strength of the constraint. If *strength* is not specified, it defaults to the value of `*default-input-strength*`, initially `:strong`.

When either `with-stays` or `with-slots-set` is executed, each of the specified constraints are applied to the slots (in an unspecified order), then the forms in *forms* are evaluated, then the constraints are removed. The value returned by the last form in *forms* is returned by the `with-stays` or `with-slots-set` form. These macros call `unwind-protect` so the constraints are removed even if an error occurs in the evaluation of the forms.

Note that these forms do not guarantee that the constraints will be satisfied during the evaluation of the forms. They may be overridden by stronger constraints.

Examples:

```
(with-stays ((recta :left)
             ((g-value recta :next) :left :weak))
         (s-value recta :right 200))
```

The above form adds a `:strong` stay constraint to the `:left` slot of `recta`, and a `:weak` stay to the `:left` slot of whatever object is stored in the `:next` slot of `recta`, evaluates the form `(s-value rectb :left 200)`, and then removes the stay constraints.

Currently, the `with-stays` or `with-slots-set` macros do not allow specifying a indirect reference path via a series of slots to follow to find the slot being constrained. It is necessary to explicitly find the object containing the slot, using `g-value`, to add a constraint to such a slot.

```
(with-slots-set ((recta :left 50 :required)
                 (recta :right 200 :required))
         (s-value recta :cn2 (m-constraint :required (left right width)
                                   (setf right (+ left width))
                                   (setf left (- right width))
                                   (setf width (- right left))))
         )
```

The above example shows how the `with-slots-set` macro might actually be used. One important use for temporary constraints is to control which slots are changed when other constraints are initially added. In the above example, we want to add a constraint between the `:left`, `:right` and `:width` slots of `recta`. When this constraint is added, Multi-Garnet may try to maintain it by executing any of the three methods, to calculate any one of these three slots from the values of the other two. The `with-slots-set` form is used in this case to set the `:left` and `:right` slots to the specified values, and ensure that the `:width` slot is initially computed when the new constraint is added.

2.8 Indirect Reference Paths

Multi-Garnet constraints specify each constrained slot by means of an "indirect reference path," a sequence of slot names. An indirect reference path may simply specify a single local slot in the "root object" containing the constraint, or may specify a sequence of slots leading to the final slot to be constrained. Multi-Garnet implements constraints with indirect reference paths by transforming them into constraints with fixed references to the final slots specified by the indirect reference paths. If any of the slots along an indirect reference path are changed, the associated fixed-reference constraint is removed from the network, this constraint is changed to refer to the new slots specified by the indirect reference paths, and the constraint is added to the network to constrain these new slots. If any of the indirect reference paths of a constraint is "broken," (the path cannot be followed because one of the slots along the path does not contain a legitimate Garnet object), then the constraint is not considered part of the constraint network.

Multi-Garnet implements indirect reference paths by distinguishing between *active* and *inactive* constraints. When a constraint is created and stored in a slot, its indirect reference paths are

followed to determine which slots are being constrained. If none of the paths is broken, then the constraint is activated, and added to the constraint network. If a slot accessed along an indirect reference path changes, (1) all active constraints with an indirect reference path using that slot are removed from the constraint network and deactivated, (2) the indirect reference paths for all inactive constraints whose paths access that slot are re-evaluated, and (3) any of these constraints whose paths are now unbroken are activated and added to the constraint network. This scheme will automatically remove constraints when their indirect reference paths are broken, and add them again whenever the paths are re-connected.

The order in which the inactive constraint paths are evaluated and the constraints are added is deliberately left unspecified. If the order matters, this indicates that there are multiple possible ways to solve the constraint network, and the user may want to alter the constraint network to select a single desired solution.

Slots along an indirect reference path can be constrained by other constraints. When constraint propagation causes such a slot to be changed, constraint propagation is completed using the existing constraint network, then any constraints whose indirect reference paths have changed are updated as described above. This may cause further constraint propagation, and indirect reference path invalidation, etc. One can thus have multiple levels of constraint networks each setting indirect reference path slots for constraints on the level below it. Note that it is possible that a constraint that will be removed (because a slot on its indirect reference path has changed) can still influence the constraint graph, and can still be used to propagate variable values, up until the moment it is removed.

Within this system, it is possible to cause infinite loops, by creating constraints that directly or indirectly set their own indirect reference path slots. For example, consider the code:

```
(create-instance 'bar nil)
(create-instance 'baz nil)
(s-value bar :b baz)
(s-value baz :b bar)
(create-instance 'foo nil
                 (:a bar)
                 (:c (m-constraint :strong ((yy (gvl :a))
                                            (xx (gvl :a :b)))
                                   (setf yy xx)))
                 )
```

When the constraint is created, it will set `foo`'s `:a` slot to `baz`, which will cause the constraint to be removed and added, which will set `foo`'s `:a` slot to `bar`, and back and forth forever.

It is very unlikely that infinite loops such as this would occur in normal Multi-Garnet programs. Still, if such a loop did occur, it would be difficult to debug the program. To help detect such loops, Multi-Garnet imposes an arbitrary limit on the number of times that invalid indirect reference paths are processed as the result of a single value propagation. When this limit is reached, the list of invalidated indirect reference paths is cleared, leaving the constraint network in a strange state where some constraints will not actually constrain the slots specified by their indirect reference paths. This is admittedly an ad-hoc solution to this (rare) problem. In a future version of Multi-Garnet it may be possible to detect such loops by analyzing the interactions of the constraint network and the indirect reference paths.

The loop-termination mechanism is controlled by the following global variables:

`*max-path-updates*` [*Variable*]

Value specifies the number of times that invalid indirect reference paths will be processed as the result of a single value propagation. Initially 10.

`*max-path-updates-warning*` [*Variable*]

Value determines whether warnings are printed when indirect reference paths are processed more than `*max-path-updates*` times. If the value is `NIL`, nothing is printed. If the value is `:error`, a continuable error is signaled. If the value is `T`, a warning message is printed. Initially `T`.

2.9 Inheriting Constraints, Indirect Path Slots, and Variable Slots

Multi-Garnet implements inheritance of multi-way constraints and constrained slots slightly differently than Garnet handles inheritance of formulas. First, constraints are only inherited when an object is created, by copying any constraints in parent slots down to the child. After a child is created, adding or removing constraints from the parent has no effect on the child. In particular, if a constraint is set to a slot in a parent, it is not automatically copied down to all children. Second, any inherited slots accessed while following indirect reference paths (including the final slot to be constrained) are copied down when they are first accessed.

After an object instance is created and initialized, any constraints in the slots of the parent instance are copied down to the child object (unless the child object overrides the slot value). These constraint copies are added to the child object using `s-value` (in an unspecified order), so they are activated if their indirect reference paths are unbroken. Note that the child's copy of a constraint is completely independent of the parent's copy of the constraint. The effect is as if a copy of the original `m-constraint` definition form had been used to create a new constraint, and it had been assigned to the child object's slot.

Note: A constraint in a parent object slot is only copied down and activated if the object and slot are that constraint's root object and slot (see Section 2.5).

If constraint variable slots and indirect reference path slots are inherited, their value is copied down when they are first accessed while following the indirect reference paths.

2.9.1 The Duplicate Constraint Problem

If a constraint is inherited, and all of its indirect reference paths slots are inherited, then it will define a duplicate constraint. For example, consider the situation:

```
(create-instance 'foo nil (:a 5))
(create-instance 'bar nil (:b 6))
(create-instance 'baz nil
                 (:obj1 foo)
                 (:obj2 bar)
                 (:cn (m-constraint :required ((a (gvl :obj1 :a))
                                               (b (gvl :obj2 :b)))
                                    (setf a b)
                                    (setf b a))))
```

The constraint in `baz`'s `:cn` slot ensures that `foo`'s `:a` slot is equal to `bar`'s `:b` slot. However, suppose that now a child instance is made of `baz` via `(create-instance 'baz2 baz)`. The constraint `:cn` will be copied down to `baz2`, and the contents of slots `:obj1` and `:obj2` will be inherited, so the new constraint in `baz2`'s `:cn` slot will also try to equate `foo`'s `:a` slot and `bar`'s `:b` slot. This will cause a cycle, which the current system cannot handle correctly. It seems unfortunate that one can get into problems simply by making an instance of an object, without overriding any slots.

It might be possible to extend Multi-Garnet to handle such duplicate constraints specially, perhaps by changing `create-instance` to inactivate duplicate constraints automatically. However, this would change the semantics of inheritance. Currently, it seems reasonable to consider duplicate constraints as programming errors.

2.10 Interactors and Multi-Garnet Constraints

Garnet interactors operate by setting specified slots in graphic objects (like feedback objects), and formulas are used to distribute this information to other parts of the user interface. If these slots are constrained by Multi-Garnet constraints, or are indirect reference path slots, setting these slots will trigger Multi-Garnet constraint propagation and/or changes in the constraint network. Therefore, in most situations, normal Garnet interactors can be used with Multi-Garnet, without change.

One exception is interactors which use destructive operations to change data structures. For efficiency, `inter:move-grow-interactor` and `inter:two-point-interactor` reuse the list structures stored in the `:box` slot of the item (or feedback object) being dragged. Since `s-value` isn't used, Multi-Garnet value propagation isn't triggered. In addition, Multi-Garnet stays are not able to prevent the value from being changed. Note that Garnet formulas have the same problem: These interactors have to call `mark-as-changed` to invalidate any formulas depending on the `:box` slot, when they destructively change this list.

This problem was handled in the current Multi-Garnet system by changing the interactors so they will check if the `:box` slot is constrained by a Multi-Garnet constraint. If it is not constrained, then the normal destructive list operation is done. If it is constrained, then the `:box` slot is set to a copy of the new points list (using an input constraint with strength `*default-input-strength*`). Presumably, the cases where this happens are few enough that they can be handled individually.

Since setting a value to a constrained slot is implemented by adding and removing a constraint, this can cause unexpected behavior with some interactors. For example, if a weak constraint equates the value in a `:box` slot to a particular value, then dragging the object with `move-grow-interactor` will not change the position of the object, because after each time the interactor sets the value (adding and then removing a strong constraint), the weak constraint will set the value back to the original value. If (`opal:update`) is called after each constraint operation, then the object will appear to bounce between the original and new positions, as the weak constraint and the interactor alternatively determine its position.

2.11 Coexistence of Garnet and Multi-Garnet

In implementing Multi-Garnet, some of Garnet's internal functions were modified to detect and handle Multi-Garnet constraints, while taking care not to remove any of the formula handling code. Regular Garnet applications that use formulas can be loaded and run in Multi-Garnet, coexisting with applications built entirely using Multi-Garnet constraints.

Garnet formulas and Multi-Garnet constraints can also be mixed in the same application, although all possible combinations are not supported. Specifically, it always possible for Garnet formulas to read the values of slots constrained by Multi-Garnet constraints, but it is somewhat more problematic for Multi-Garnet constraints to constrain slots that are set by Garnet formulas. This is a difficult situation because of the different evaluation styles used by formulas (lazy evaluation) and Multi-Garnet constraints (eager evaluation), as well as the desire to respect Multi-Garnet constraints. For example, if a slot is constrained by a required stay, it should not be changed, even if it has an associated formula that is invalidated.

Currently, formulas associated with constrained slots are handled as follows: When such a formula is invalidated, an input constraint (with strength `*formula-set-strength*`, normally `:strong`) is created and temporarily added and removed to set the slot. If the input constraint is strong enough, the formula is evaluated, and the new value is propagated through the network. If the input constraint is overridden by a stronger constraint, the formula is marked as valid (even though it is not evaluated).

The situation is actually more complicated, since formula evaluation may cause constraint propagation, which may cause other formulas to be invalidated. It is possible to created loops between formulas and Multi-Garnet constraints. Formula evaluation is handled through the same mechanism used to update invalid indirect reference paths (see Section 2.8), and the variable `*max-path-updates*` is used to terminate possible infinite loops.

This mechanism allowing formulas to be associated with constrained slots is new and experimental. It was implemented primarily to allow the use of constraints with existing Garnet gadgets containing formulas. Therefore, not all possible combinations of formulas and constraints are supported. In particular, it is not possible to add a formula to a slot that is already constrained: the formula must already be associated with the slot before it is constrained. Illegal combinations are detected, and an error is signaled.

2.12 Errors under Multi-Garnet

One of the goals for Multi-Garnet was to be as robust as possible, doing the most reasonable thing rather than just crashing if an error occurs. In particular, it is very undesirable for an error to leave the constraint solver data structures in a bad state, since it would be virtually impossible to fix them up and continue. Here is how Multi-Garnet handles several different types of errors:

1. Required-required conflicts.

   When a required constraint is added to the constraint network, and it cannot be enforced because of a conflicting required constraint, the newly-added constraint is left unsatisfied (in effect, reducing its strength to a strength between `:strong` and `:required`). If at some future point the other conflicting constraint is removed, the newly-added constraint will be enforced at that time. Note that if a constraint network contains any unsatisfied required constraints, then by definition it is an unsolved network, and none of the variable values should be trusted.

   `*unsatisfied-required-constraint-warning*` [*Variable*]

   If this variable is non-`NIL`, a warning message will be printed when a required constraint is added to the constraint network, and it cannot be enforced because of a conflicting required constraint. Initially `T`.

2. Constraint cycles.

   The SkyBlue constraint solver is more tolerant of constraint cycles than the DeltaBlue solver. Whereas DeltaBlue would signal a fatal error and halt if it encountered a cycle of constraint methods, SkyBlue can handle this situation without halting. This does not mean that SkyBlue will necessarily *solve* a cycle, producing correct values for the variables in the cycle. Given a cyclic constraint network, SkyBlue may choose a set of methods containing a cycle. When such a cycle is detected during constraint propagation, SkyBlue will stop propagating values.

   `*sky-blue-cycle-warning*` [*Variable*]

   If this variable is non-`NIL`, a warning message will be printed when a cycle is detected. Initially `NIL`.

   An area for future work is adding support for more powerful solvers for solving cycles of constraints, which SkyBlue could call when such a cycle is detected.

   If there is such a cycle, then by definition the constraint network is not solved, and the variable values should not be trusted. However, if the cycle is later broken by removing constraints in the cycle, then SkyBlue will propagate values to solve the network again.

3. Illegal mixtures of formulas/constraints/variables

   Currently, there are a number of situations that Multi-Garnet cannot handle, such as installing a formula on a slot containing a Multi-Garnet constraint, or a constraint being produced as the output of a constraint method. When such a situation is detected, an error is signaled. Unfortunately, in the current implementation it may not be safe to continue from the error. For example, if a constraint method produces a constraint as its value while propagating values around the constraint network, this may leave the constraint network structures in a bad state.

4. Errors while evaluating constraint methods.

   If an error occurs while evaluating a constraint method, this can leave the system in a bad state. This was also true when evaluating formulas in Garnet.

2.13 Examining Constraints and Variables

The following functions can be used to examine the state of Multi-Garnet constraints and variables:

(`constraint-p` *constraint*) [*Function*]

Returns `T` if *constraint* is a constraint, `NIL` otherwise.

(`constraint-state` *constraint*) [*Function*]

Returns four multiple values: *connection*, *enforced*, *object*, and *slot*.

connection is one of the following keywords: `:unconnected`, `connected`, `:graph`, or `:broken-path`. If *connection* is `:unconnected`, this means that the constraint does not have a root object, and would be activated if it was stored in an object slot. If *connection* is `:connected`, this means that the constraint's indirect reference paths have been successfully resolved, but it has not been added to the constraint graph yet (this is a transitional state that should not be seen in normal circumstances). If *connection* is `:graph`, the constraint is part of the constraint graph. If *connection* is `:broken-path`, one of the indirect reference paths of the constraint is broken, so the constraint is not part of the constraint graph.

enforced is `T` if the constraint is currently an enforced constraint in the constraint graph, `NIL` otherwise. It should be enforced if it is in the graph, unless the constraint is overridden by a stronger constraint.

object is the root object used when interpreting this constraint's indirect reference paths. If this is `NIL`, then the constraint is not stored in any object. Usually, such constraints are `:unconnected`, though there are some constraints, such as the input constraints used to implement `s-value`, that do not have a root object.

slot is the slot in the root object containing this constraint.

(`variable-state` *object slot*) [*Function*]

Returns three multiple values: *var-p*, *valid*, and *path-slot-p*. *var-p* is `T` if the specified object and slot is used as a Multi-Garnet variable, `NIL` otherwise. *valid* is `T` if the variable value is valid, `NIL` otherwise. *path-slot-p* is `T` if the specified object and slot is used as a slot along a Multi-Garnet indirect reference path, `NIL` otherwise.

Currently, the only situation where a variable will be marked invalid is when it is "downstream" of a cycle of methods. In a future version of Multi-Garnet, each variable will be marked valid if and only if its value is part of a correct solution to the constraint network (according to the hierarchical constraint theory). In this case, if there is a cycle in the graph that SkyBlue cannot solve, or there are two conflicting required constraints that cannot both be satisfied, then all variables in the graph would be marked as invalid.

2.14 Creating and Using Plans

When the SkyBlue constraint solver adds or removes a constraint from the network, it performs separate planning and execution operations to resatisfy the constraints. During the planning stage, methods from the constraints are chosen and ordered; during the execution stage, these methods

are actually executed to compute new values for the variables that will satisfy the hierarchy of constraints. In many interactive graphics applications, new input values is repeatedly fed into the same network of constraints, repeatedly executing the same sequence of methods. For example, when dragging an object using `inter:move-grow-interactor`, the `:box` slot of the object being dragged will be repeatedly set, followed by the same constraint propagation. In this case, it is possible to improve performance by extracting the "plan" for the constraint propagation once, and repeatedly executing it with different values for the `:box` slot.

Multi-Garnet provides functions for extracting and reusing plans, and they can be used to improve performance in some situations, but they should be used very carefully. A given plan is derived from a given constraint network, and only has meaning for that set of constraints. If a constraint is added or removed from the part of the network containing the constraints in the plan, this would render the plan invalid, and the plan should not be executed. Multi-Garnet detects when a plan is invalid, and refuses to execute invalid plans.

(`create-plan` *constraints*) [*Function*]

Given a list of constraints, creates and returns a plan that will propagate values starting with these constraints. Any constraints that are not currently enforced are ignored.

(`valid-plan-p` *plan*) [*Function*]

Returns `T` if *plan* is currently valid, otherwise `NIL`.

(`run-plan` *plan*) [*Function*]

Executes the plan, if it is valid. If the plan is invalid, this signals a continuable error; continuing from the error just returns from `run-plan` without executing the plan. Note that executing a plan may cause indirect reference paths or formulas on constrained slots to be invalidated, which may cause other constraints to be added and removed, which may invalidate the plan.

(`propagate-plan-from-cn` *constraint*) [*Function*]

This is a convenient function for generating and using a plan based on a single constraint. This extracts a plan starting from the constraint, caches it, and runs it. If there is already a cached plan that is valid, then this plan is used instead. In the typical case, when the constraint network is not being changed between calls to this function, a plan will be generated on the first call, and reused on the remaining calls. However, if the constraint network is being changed between calls, then this will repeatedly extract and run a plan, which may be slower than simply repeatedly adding constraints to inject new values into the network.

Here is an example showing how the plan functions can be used. Suppose that one wanted a version of `inter:move-grow-interactor` that reused a plan to repeatedly set the `:box` slot of an object being dragged. This could be done via:

```
(create-instance
 'my-inter inter:move-grow-interactor
 (:start-action
   #'(lambda (inter obj pts)
       ;; set up external input value
       (setq *inter-box* (copy-list pts))
       ;; add input constraint to set :box of obj
       (s-value obj :inter-box-cn
                (m-constraint :strong (box) (setf box *inter-box*)))
       ))
 (:running-action
   #'(lambda (inter obj pts)
       ;; update new box value
       (setq *inter-box* (copy-list pts))
       ;; extract and execute plan to propagate this new value
       (propagate-plan-from-cn (g-value obj :inter-box-cn))
       ))
 (:stop-action
  #'(lambda (inter obj pts)
      ;; remove plan constraint
      (s-value obj :inter-box-cn nil)))
 ...)
```

Note that the constraint added to set the `:box` slot, and to be the head of the plan, takes its value from the global variable `*inter-box*`, external to the constraint network. This is important. If the new value was accessed through the normal constraint slot access mechanisms, via a constraint such as `(m-constraint :strong (box new-val) (setf box new-val))`, then any attempt to set the `:new-val` slot would lead to adding an input constraint, and constraint propagation, and the plan would be invalidated.

2.15 Enabling and Disabling Multi-Garnet

In implementing Multi-Garnet, a number of internal Garnet system functions were modified to recognize and handle Multi-Garnet constraints. When Multi-Garnet is loaded, the old versions of these functions are saved, and new versions are installed. These modifications can be reversed using the following functions. Under normal circumstances, it should never be necessary to call these functions.

`(enable-multi-garnet)` [*Function*]

Modifies Garnet system so that Multi-Garnet constraints can be handled. This function is called when `load-multi-garnet.lisp` or `compile-multi-garnet.lisp` is loaded.

`(disable-multi-garnet)` [*Function*]

Restores the original Garnet function definitions. While Multi-Garnet is disabled, any Multi-Garnet manipulations, such as adding constraints to slots, or setting values to constrained slots, will not work correctly.

`(multi-garnet-enabled)` [*Function*]

Returns `T` if Multi-Garnet is currently enabled, `NIL` otherwise.

Chapter 3

A Scatterplot in Multi-Garnet

The file `scatterplot.lisp` in the Multi-Garnet v2.1 release contains a large Multi-Garnet example: a scatterplot displaying a set of points. Multi-Garnet constraints are used to specify relationships between the data values, the screen positions of the points, and the positions of the X and Y-axes. As the scatterplot points and axes are manipulated, Multi-Garnet maintains the constraints by propagating values through the constraint network. The exact behavior of an interaction is controlled by the current interaction mode, which "anchors" values that should not be changed when satisfying the constraints.

This scatterplot program could have been written entirely in Garnet, using one-way formulas to connect the different elements, but it would not have been easy. Each different interaction mode would have to enable exactly the right set of formulas to propagate the values correctly. Adding a new interaction mode would require examining the entire network of formulas. In contrast, the Multi-Garnet scatterplot program declares the relationships between the scatterplot elements once, and uses the Multi-Garnet constraint solver to choose which methods to execute to maintain the constraints. New interaction modes are specified by adding a few stay constraints to control constraint propagation.

To try this program, compile and load the file `scatterplot.lisp`. Once this file is loaded, execute `(user::create-scatterplot-demo)` to create the window displaying the scatterplot. Depending on the current interaction mode, dragging the scatterplot points and the axes will produce different results. The current interaction mode is specified via the buttons on the left of the window.

This document describes how to operate this example, and explains how the different interaction modes are implemented using multi-way hierarchical constraints. The file `scatterplot.lisp` includes many comments explaining details of the implementation.

3.1 The Scatterplot Window

Figure 3.1 shows the initial state of the scatterplot window. It contains two scatterplots displaying various slots of a single collection of data objects. The left scatterplot displays the `:a` and `:b` slots of the objects, and the right scatterplot displays the `:c` and `:a` slots.

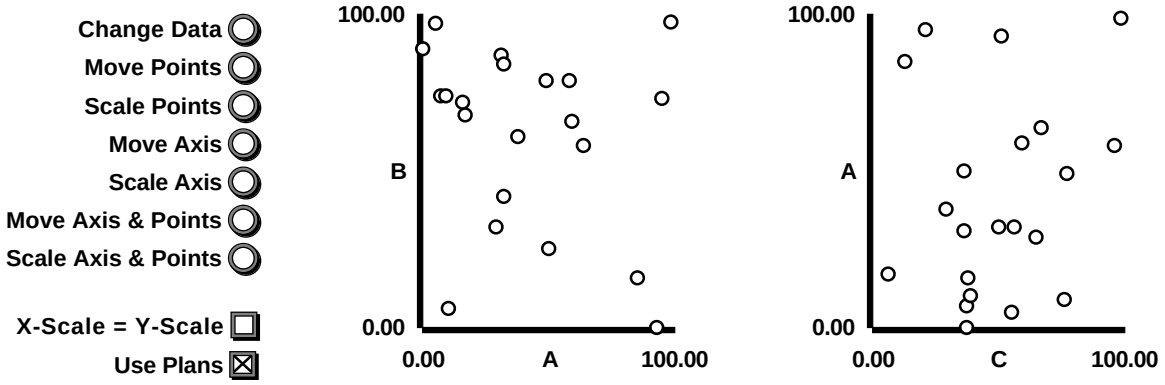


Figure 3.1: Initial Scatterplot Window

The scatterplot uses Multi-Garnet constraints to maintain relationships between the object fields, the positions of points in the scatterplot, and the positions and ranges of the scatterplot axes. Figure 3.2 shows the same window, after the left scatterplot points and axes have been moved and resized. This is still a meaningful scatterplot, displaying the same data.

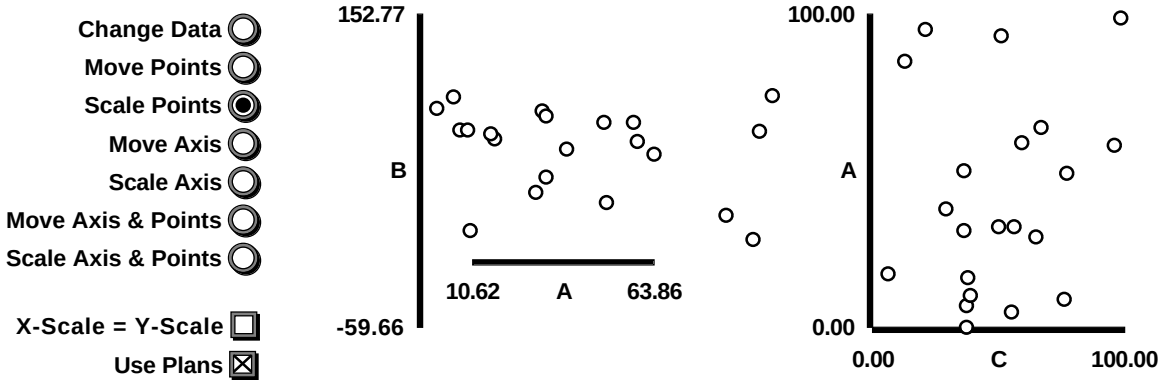


Figure 3.2: Scatterplot After Changes to Points and Axes

The buttons on the left control the interactors used to move and reshape the points and axes. At any one time, only the single interaction mode specified by the radio buttons is enabled. A real data analysis system would probably link different interaction modes to different control-key and mouse-button combinations. For the demo, it is less confusing to specify the interaction mode explicitly.

The different interaction modes specify different ways that the various parts of the scatterplot should be modified as the points or axis bars are moved. For example, in the "Change Data" mode, moving a point will actually change the data object being represented, without changing any other parts of the scatterplot. On the other hand, in the "Move Points" mode, moving a point will drag all of the scatterplot points the same distance, updating the axis range numbers appropriately. For any given interaction (dragging a point, or reshaping an axis), there are multiple possible ways that the constraints between the parts of the scatterplot can be maintained. The different interaction modes demonstrate some of these possibilities.

3.2 The "Use Plans" Button

The button labeled "Use Plans" in the lower left of the scatterplot window controls whether the interactors will construct and reuse constraint plans during an interaction. It is initially selected, so plans are used. The speed of interactions with and without the use of plans can be compared by switching the "Use Plans" button off. This does not change the behavior of the interactions in any way.

3.3 The Scatterplot Constraint Network

Figure 3.3 displays the network of constraints and slots used to relate the position and range of the left scatterplot's x-axis and y-axis to the position of a *single* point, the one in the upper-right corner. In the network diagram, the black boxes represent constraints, and the white ones represent variables. The arrows indicate the input and output variables of the constraint method used to maintain each constraint. All of the constraints have required strength, unless marked otherwise. The point's `:box-cn` constraint relates its `:box` slot, specifying the size and position of the circle, to the `:x` and `:y` slots, containing the coordinates of the center of the circle. The `:x-cn` constraint relates the point's `:world-x` slot, containing the data value being measured by the x-axis, to the point's x-coordinate, according to the values of the x-axis slots `:offset` and `:scale`. This is a multi-way constraint, which can calculate any of the four slot values, given the other three. Likewise, the `:y-cn` constraint relates the point's `:world-y` slot to the point's y-coordinate and the y-axis `:offset` and `:scale`.

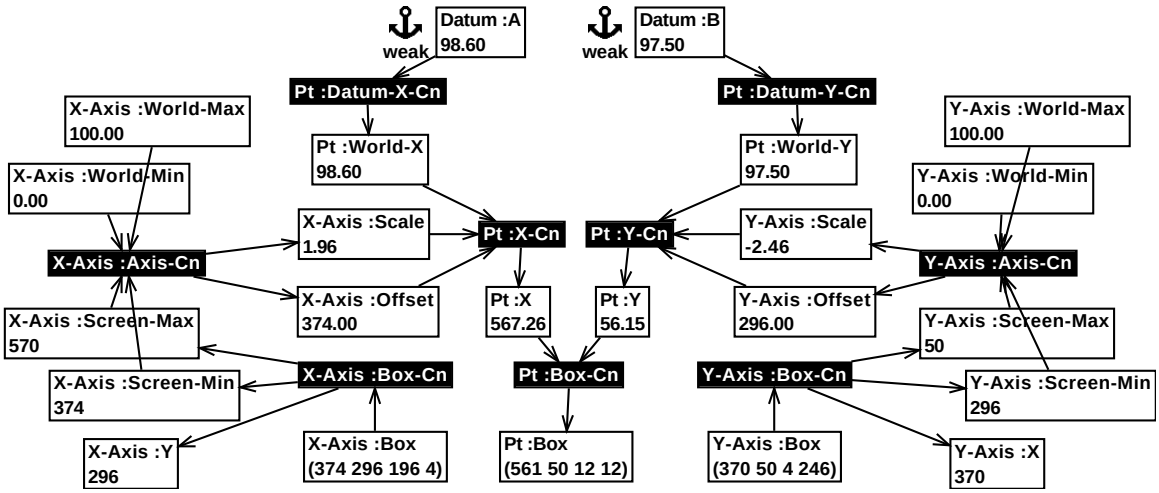


Figure 3.3: Initial Scatterplot Constraint Network

`:world-x` and `:world-y` are derived from the datum object being represented by this point. In this case, the connection between the datum and these slots is specified by the `:datum-x-cn` and `:datum-y-cn` constraints, which equate these values to the `:a` and `:b` slots of the datum object respectively. These constraints could be more complex, calculating values using multiple slots of the datum objects. The anchor symbols specify that the datum `:a` and `:b` slots are constrained by

weak stay constraints. These weak stays prevent the data slot values from being changed, if this is possible without violating stronger constraints.

The x-axis `:axis-cn` constraint maintains the relationship between the x-coordinates of the ends of the axis (`:screen-min` and `:screen-max`), the data values at the ends of the axis (`:world-min` and `:world-max`), and the `:scale` and `:offset` slots used to position points relative to the axis. This constraint defines multi-output methods to calculate the values of each of these three pairs of slots from the values of the other two pairs. Although the two ends are labeled "min" and "max", this is just a convenient labeling. `:screen-max` can be less than `:screen-min`, as in the case of the y-axis, where the y-coordinates increase downwards. `:world-max` can also be less than `:world-min`, if the data values are flipped around an axis.

Finally, the x-axis `:box-cn` constraint packs the values of the `:screen-min` and `:screen-max` slots, along with the x-axis `:y` slot determining the y-coordinate of the axis, into the x-axis `:box` slot, determining the size and position of the x-axis.

The y-axis `:axis-cn` and `:box-cn` constraints are similar, except that the `:screen-min` and `:screen-max` slots contain the y-coordinates of the ends of the axis, and the y-axis `:x` slot holds the x-coordinate of the axis.

Given this constraint network, if any of the slots are changed, Multi-Garnet will maintain the constraints by altering the values of other slots. However, there may be many possible ways to satisfy the constraints. If a point's `:x` slot is changed, the `:x-cn` constraint could be satisfied by changing the x-axis `:scale` or `:offset` slots, or the point's `:world-x` slot (though this last option would not be chosen unless some stronger constraint overrode the `:weak` stay on the datum `:a` slot). One can restrict the propagation paths chosen by adding additional stay constraints to slots. By anchoring different combinations of slots, one can produce a number of different behaviors.

3.4 The "Change Data" Interaction Mode

When the "Change Data" interaction mode is selected, dragging a point with the left mouse button will actually change the datum being represented by the scatterplot point, without changing the axes or the other data points. When such a point is changed in the left scatterplot, changing the `:a` and `:b` slots of a datum object, it will also change the position of the corresponding point in the right scatterplot. Because the right scatterplot displays the `:a` slot on the *vertical* axis, moving a point horizontally in the left axis will move it vertically in the right axis.

Figure 3.4 shows the constraint network while this interaction is proceeding. The "circle and triangle" symbol next to the "Pt :Box" box specifies that there is a strong input constraint changing the value of the `:box` slot as the Garnet interactor moves this point. The thick arrows indicate the "data flow" from the `:box` slot of the point being moved to the datum object slots actually being changed. Other constraints (not shown) use the value of the `:a` data field to position the corresponding point in the right scatterplot.

While this interactor is running, stay constraints are added to other variables to control how the constraints are maintained. During this interaction, required stay constraints are added to the `:scale` and `offset` slots of the x and y-axes, so that neither of the axes will be changed during the interaction. Since all of the other points in the scatterplot are positioned according to these slots, they won't move either.

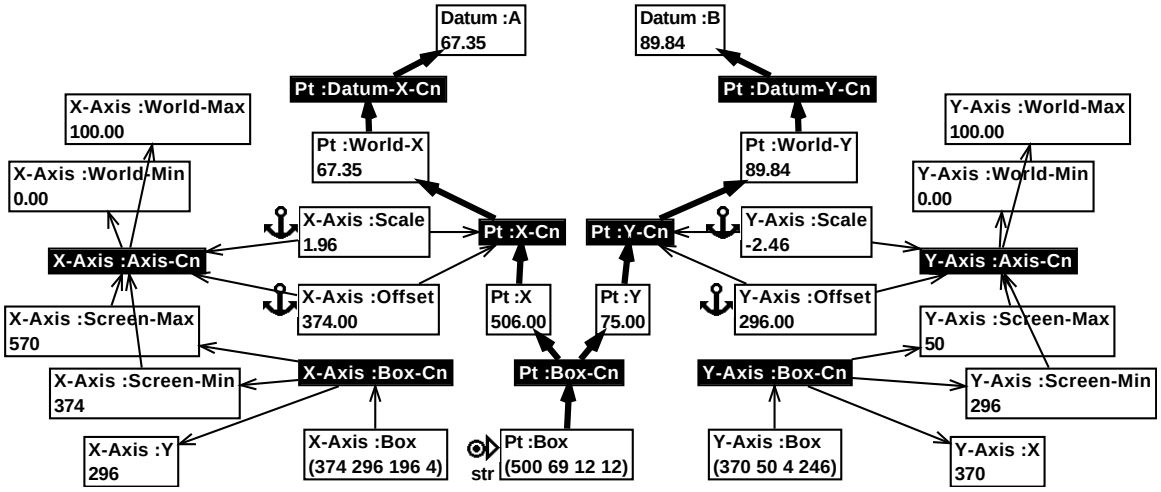


Figure 3.4: Scatterplot and Constraint Network During "Change Data" Interaction Mode

Note: If an anchor is displayed without a strength, its strength is required. An anchor symbol is only displayed if a stay constraint is currently anchoring the specified variable. In Figure 3.4, `:datum-x-cn` is determining the value of the datum `:a` slot, so the anchor symbol for its weak stay constraint is not displayed.

3.5 The "Move Points" Interaction Mode

When the "Move Points" interaction mode is selected, dragging a point with the left mouse button will drag all of the other points in the scatterplot along with it, maintaining their relative positions (see Figure 3.5a). The axes will not move, but their range numbers will change to be consistent with the position of the data cloud.

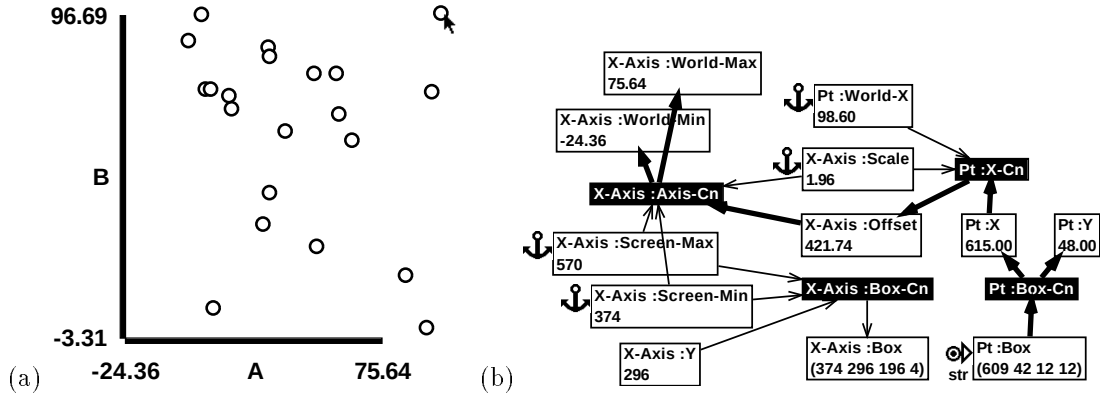


Figure 3.5: Scatterplot and Constraint Network During "Move Points" Interaction Mode

Figure 3.5b shows the constraint network when this interaction is proceeding. Only the x-axis half

of the network is shown: the y-axis network is symmetrical. Also the datum object slots are not shown: in this and the following network diagrams, the datum objects are not changed.

While this interactor is running, required stay constraints are added to `:world-x`, so the data value will not change, and to the x-axis `:scale`, so that the scale used to determine relative x-distances does not change. Required constraints are also added to the axis `:screen-max` and `:screen-min` slots, so the axis positions will not change. The only way to satisfy the constraints when the `:box` slot changes is to modify the `:offset` slot, and then use this to modify the axis range numbers. As the `:offset` slot is changed, this causes all of the points in the scatterplot to be moved by the same amount, so their relative positions stay the same.

3.6 The "Scale Points" Interaction Mode

When the "Scale Points" interaction mode is selected, dragging a point with the left mouse button will stretch the point cloud, scaling the point positions independently in both the x and y directions (see Figure 3.6a). The axes will not move, but their range numbers will change to be consistent with the position and size of the point cloud.

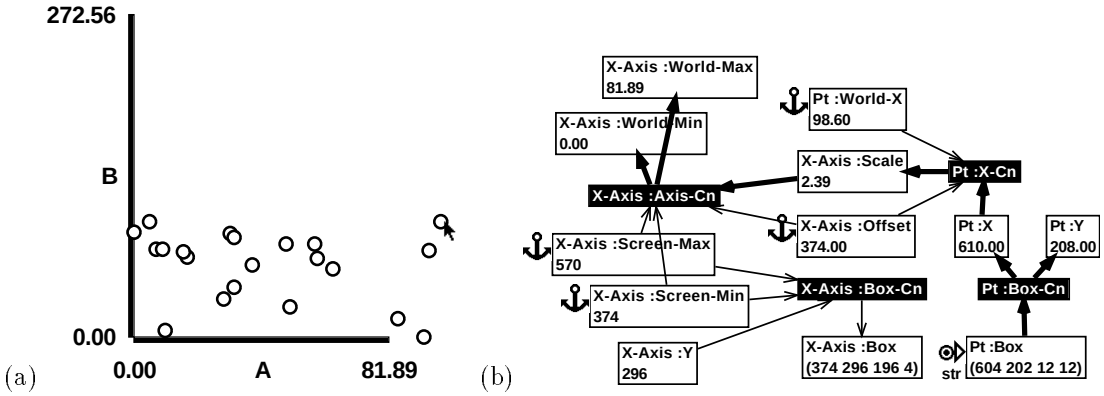


Figure 3.6: Scatterplot and Constraint Network During "Scale Points" Interaction Mode

The scaling is done relative to the origin of the scatterplot (where `:world-x` and `:world-y` would both equal 0). When the selected point is dragged to this position, the point cloud shrinks down to a single point. The selected point can even be dragged to the "other side" of the origin, causing the point cloud to be reversed.

Figure 3.6b shows the constraint network when this interaction is proceeding. It is similar to Figure 3.5b, except that the anchor is on the `:offset` slot rather than the `:scale` slot.

3.7 The "Move Axis" Interaction Mode

When the "Move Axis" interaction mode is selected, dragging one of the axis bars with the left mouse button will move it, without changing the scatterplot points or the other axis (see Figure 3.7a). The

axis range numbers are changed to be consistent with the position of the axis relative to the data cloud. This allows the axis to be used as a "ruler" to examine the position of particular points.

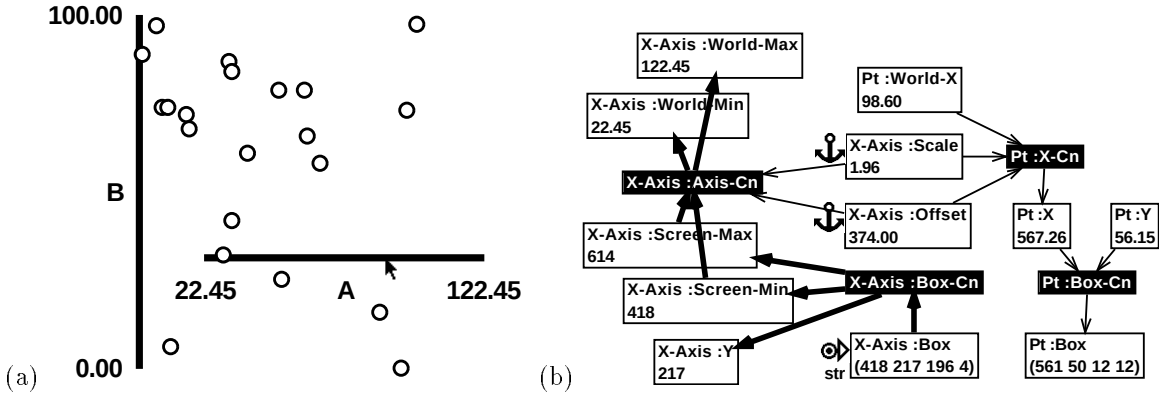


Figure 3.7: Scatterplot and Constraint Network During "Move Axis" Interaction Mode

Figure 3.7b shows the constraint network when this interaction is proceeding. While this interactor is running, required stay constraints are added to the `:scale` and `offset` slots of the x-axis, so that the point positions will not be changed during the interaction. As the `:box` slot of the x-axis is changed by the Garnet interactor, it is unpacked by the `:box-cn` constraint, and then these values are used to update the axis range numbers.

3.8 The "Scale Axis" Interaction Mode

When the "Scale Axis" interaction mode is selected, clicking and dragging on one of the axis bars with the left mouse button will stretch or shrink the axis, relative to the far end (see Figure 3.8a). The axis range numbers are changed to be consistent with the position and size of the axis relative to the data cloud. The scatterplot points and the other axis are not changed.

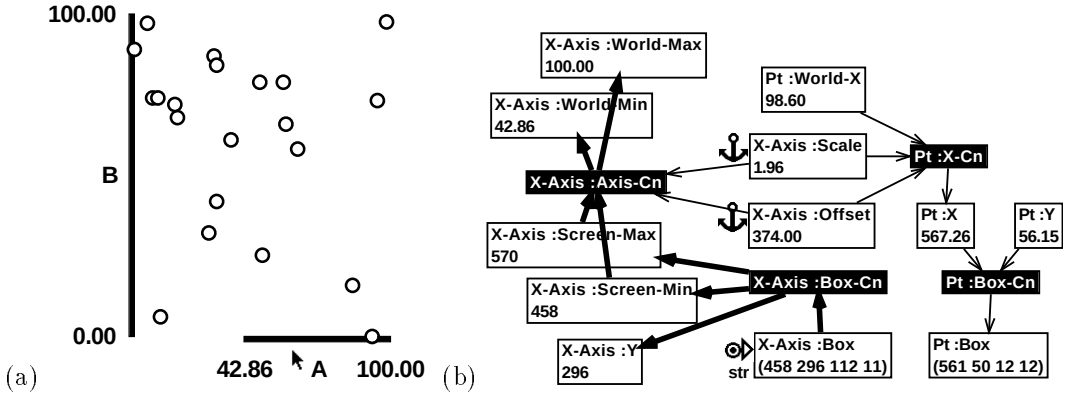


Figure 3.8: Scatterplot and Constraint Network During "Scale Axis" Interaction Mode

It is possible to flip an axis by reshaping it so its ends are reversed. The axis range numbers will be

calculated correctly in this case. However, the axis rectangle will disappear, and warning messages will be printed by the Garnet display routines, because the axis rectangle will have a negative width (or height).

Figure 3.8b shows the constraint network when this interaction is proceeding. This is almost exactly the same as the network during the "Move Axis" interaction mode (Figure 3.7b). Required stay constraints are added to the `:scale` and `offset` slots of the x-axis, so that the point positions will not be changed during the interaction, and the changed values of the `:box` slot propagate to change the axis range numbers. The only difference between these two interactions is that the `move-grow-interactor` used for the "Move Axis" interaction mode has its `:grow-p` slot set to `NIL`, so it responds to mouse movements by setting the axis' `:box` slot to simply move the axis rectangle. The `move-grow-interactor` used for the "Scale Axis" interaction mode has its `:grow-p` slot set to `T`, so it responds to mouse movements by setting by setting the axis' `:box` slot to correspond to reshaping the axis rectangle.

Sometimes the `move-grow-interactor` used for the "Scale Axis" interaction mode will set the fourth number in the `:box` list (the height) to different numbers. However, the constraints which unpack the `:box` slot into the four `:left`, `:top`, `:width`, and `:height` slots (not shown in the constraint networks here) always set the `:height` slot of a horizontal axis to a single constant value, so the axis rectangle displayed will not actually change height.

3.9 The "Move Axis & Points" Interaction Mode

When the "Move Axis & Points" interaction mode is selected, dragging one of the axis bars with the left mouse button will move it along with the scatterplot points (see Figure 3.9a). The axis' range numbers are not changed, since it is still measuring the same range relative to the scatterplot points. The other axis is unchanged.

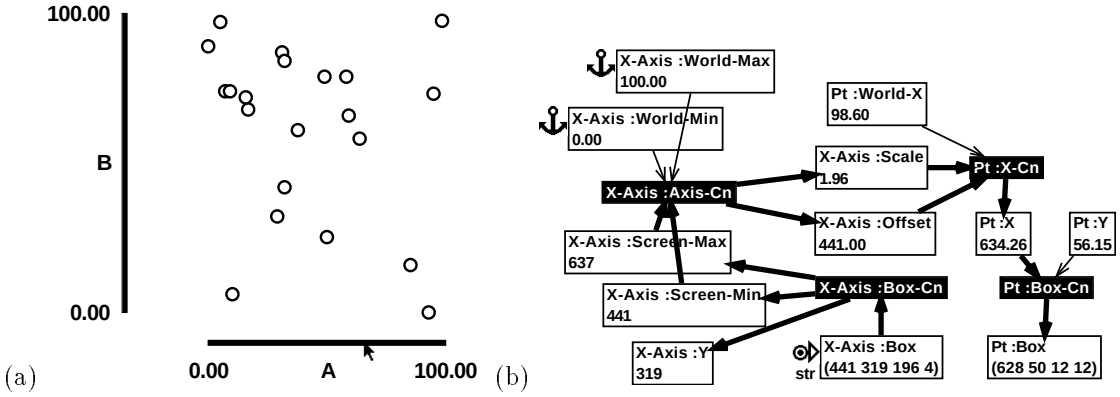


Figure 3.9: Scatterplot and Constraint Network During "Move Axis & Points Interaction Mode

One use for this mode is to allow combining multiple scatterplots. For example, in Figure 3.10, the "C" axis and the points of the right scatterplot have been dragged over to the left scatterplot. This mode would be particularly useful when trying to discover common patterns in data plotted in multiple scatterplots.

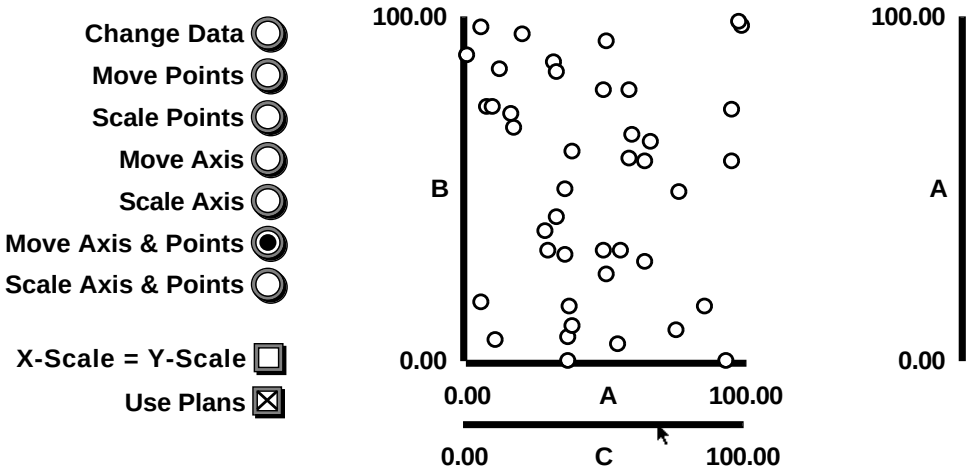


Figure 3.10: Overlaying Two Scatterplots

Figure 3.9b shows the constraint network while this interactor is running. Required stay constraints are added to the `:world-min` and `:world-max` slots of the x-axis, so that the range numbers will not change. As the axis is moved, new `:scale` and `:offset` values are calculated that correspond to the same range numbers in the new position. All of the scatterplot points use the `:world-min` and `:world-max` slots of the x-axis to calculate their x-position, so all of the points are repositioned along with the axis.

Note that the length of the axis is not changed during this interaction, so the `:scale` value will always be recalculated as the same number.

3.10 The "Scale Axis & Points" Interaction Mode

When the "Scale Axis & Points" interaction mode is selected, dragging on one of the axis bars with the left mouse button will stretch or shrink the axis, relative to the far end, along with the scatterplot points (see Figure 3.11a). The axis range numbers are not changed, since it is still measuring the same range relative to the scatterplot points. The other axis is unchanged. This can be used to adjust the scales of multiple scatterplots, to compare point clouds that may be shown with different scales. Unlike the "Scale Points" interaction mode, which changes the x and y scales independently, this just changes the scale along one axis.

Figure 3.11b shows the constraint network while this interactor is running. This is almost exactly the same as the network during the "Move Axis & Points" interaction mode (Figure 3.9b). Required stay constraints are added to the `:world-min` and `:world-max` slots of the x-axis, so that the range numbers will not change. As the axis is reshaped, new `:scale` and `:offset` values are calculated that correspond to the same range numbers for the reshaped axis. All of the scatterplot points use the `:world-min` and `:world-max` slots of the x-axis to calculate their x-position, so all of the points are repositioned to move and scale the points along with the axis.

Note that it is possible for both the `:scale` and `:offset` slots to contain new numbers as an axis is scaled. As an axis is scaled, this may move the 0.0 point along this axis, which would change the

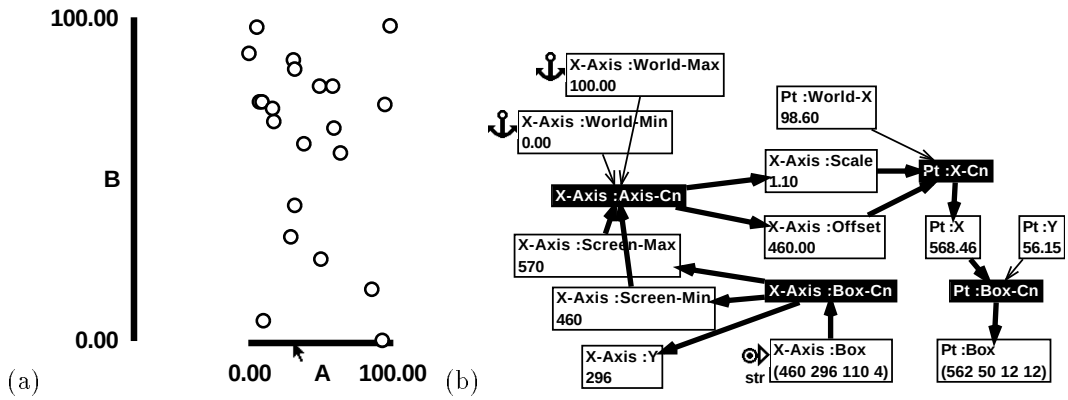


Figure 3.11: Scatterplot and Constraint Network During "Scale Axis & Points Interaction Mode

offset value.

3.11 The "X-Scale = Y-Scale" Button

By adding additional constraints to the network, the behavior of some interactions can be modified. For example, suppose that the x and y axes were measuring the same basic units, such as inches against inches. In this case, it might be desirable to restrict both axes to use the same scale in units per pixel. When the "X-Scale = Y-Scale" button is selected, a strong constraint is added equating the `:scale` slots of the X-axis and Y-axis of the left scatterplot. If these scales are not already equal, the scatterplot points may be moved when the constraint is added.

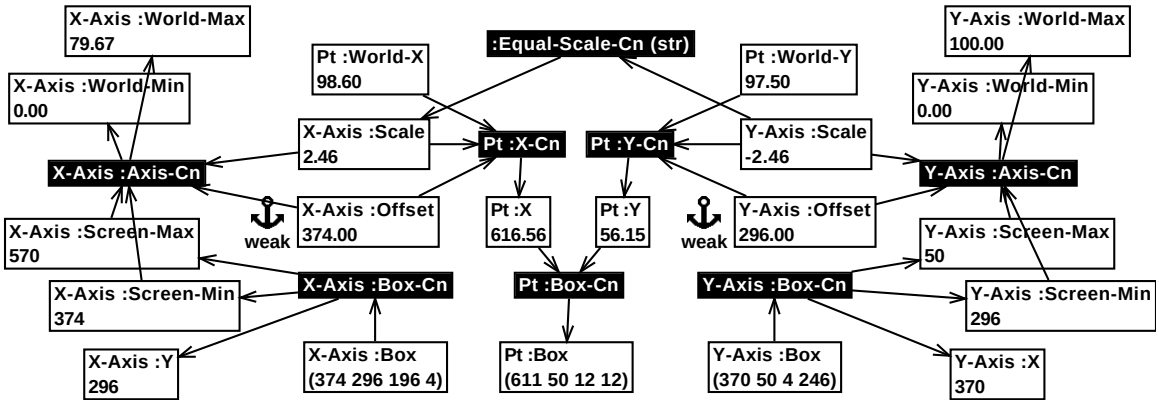


Figure 3.12: Constraint Network With "X-Scale = Y-Scale" Button Selected

Figure 3.12 shows the network when this constraint is added. When the button is selected, two weak stay constraints are also added to the `:offset` slots of the two axes, as explained below. Note that the `:equal-scale-cn` actually multiplies the y-axis scale by -1, to compensate for the y-coordinates which increase downwards (opposite from the normal direction of increasing values on a vertical axis).

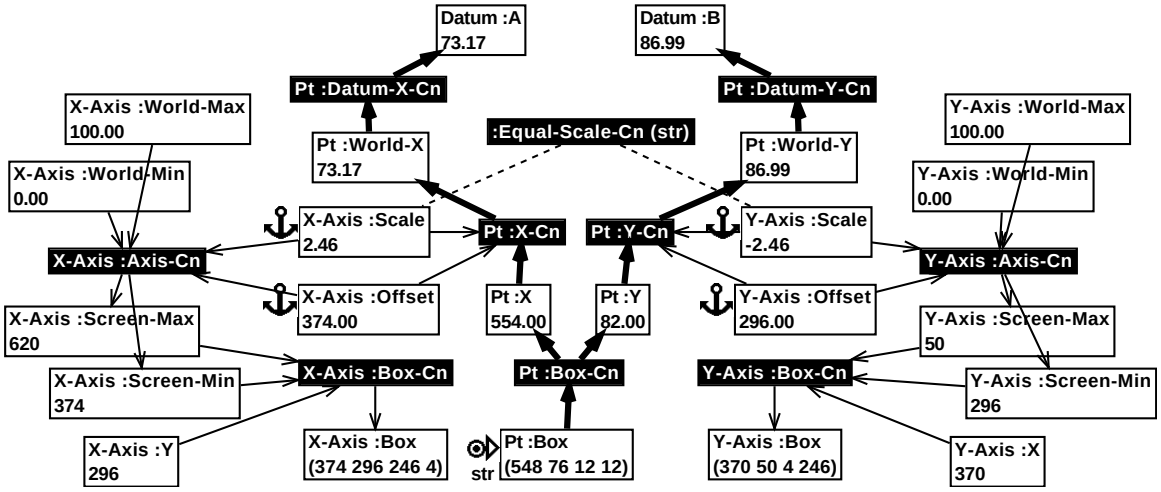


Figure 3.13: Constraint Network for "Change Data" Mode With Equal-Scale Constraint

Most of the interaction modes act the same when the scale equality constraint is added. For example, the "Change Data" mode can be used to change data values as before. Figure 3.13 shows the constraint network while this interaction is running. Because this interaction adds required stays to the X-axis and Y-axis `:scale` slots, neither of the two methods of the `:equal-scale-cn` constraint can be run (setting the x-axis scale from the y-axis scale, or vice-versa), so the constraint is unsatisfied, as indicated by the dashed lines from the box for this constraint. Actually, in this case the constraint is satisfied, because both of the scales are equal, but the current constraint system does not detect this. The `:equal-scale-cn` constraint was specified as a strong constraint, rather than a required constraint, in order to allow it to be overridden in this situation.

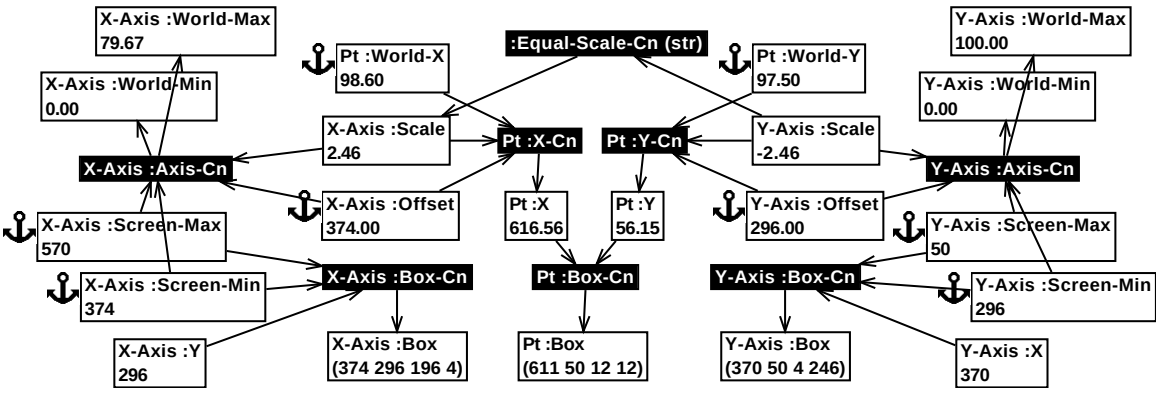


Figure 3.14: Constraint Network for "Scale Points" Mode With Equal-Scale Constraint

The "Scale Points" interaction mode is essentially disabled when the scale equality constraint is added. Figure 3.14 shows the constraint network while this interaction is running. Required stays are added to the `:world-x` and `:world-y` slots of the point being moved, and to the `:offset` slots of the x and y axes. Therefore, the only way to satisfy the point's `:x-cn` and `:y-cn` constraints is to set the x-axis and y-axis `:scale` slots. However, in order to set both of these slots, the strong

`:equal-scale-cn` constraint has to be overridden. It cannot be overridden by the strong input constraint the interactor uses to set the point's `:box` slot, so the point is not moved at all.

Another way to look at this situation is that the "Scale Points" interaction mode cannot set the x and y scales to two independent values, while the `:equal-scale-cn` constraint is ensuring that they are equal. If the `:equal-scale-cn` constraint was weaker than the input constraint, say if it was a medium constraint, the input constraint could override it during the interaction. However, when the input constraint was removed, the scale values would be changed to satisfy the `:equal-scale-cn` constraint again.

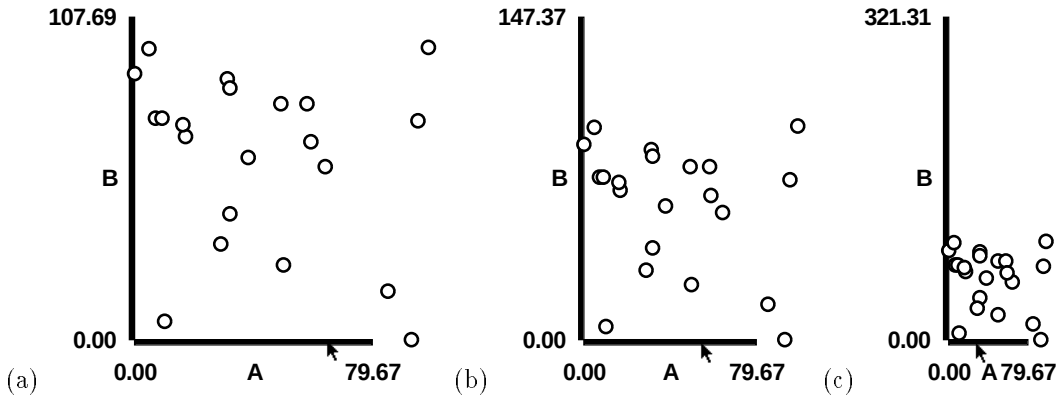


Figure 3.15: Collapsing the Point Cloud

The behavior of the "Scale Axis & Points" interaction mode is changed in an interesting way when the `:equal-scale-cn` constraint is added. As one axis is scaled, the other axis is scaled evenly, collapsing or expanding the point cloud uniformly in both directions (see Figure 3.15).

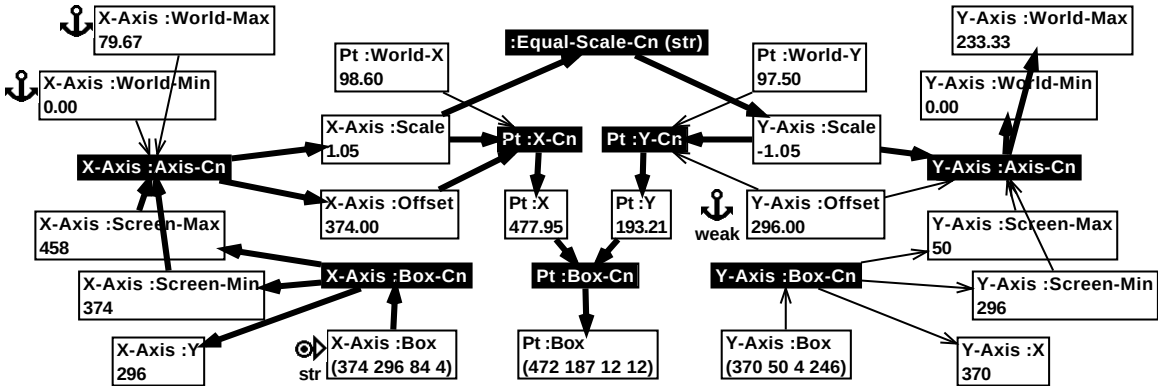


Figure 3.16: Constraint Network for "Scale Axis & Points" Mode With Equal-Scale Constraint

Figure 3.16 shows the constraint network while this interaction is running. The modified x-axis `:box` slot propagates to the x-axis `:scale` slot, which the `:equal-scale-cn` constraint copies to the y-axis `:scale`, which changes the y-axis range numbers. The changed x-axis `:scale` and `:offset` values, as well as the y-axis `:scale`, are used to reposition the scatterplot point in the network (as well as all of the other points in the scatterplot).

The weak stay on the y-axis `:offset` prevents the system from selecting a propagation path where the `:y-cn` constraint for one of the scatterplot points is satisfied by modifying the y-axis `:offset`, rather than the point's `:y` slot. If this happened, the effect would be that the rescaled point cloud would shrink and expand around the y-position of this point, rather than around the $(0, 0)$ point of the scatterplot.

There is actually more than one possible propagation path that could have been chosen with this network. It is possible for the y-axis `:axis-cn` constraint to be satisfied by changing `:screen-max` and `:screen-min`, moving and reshaping the y-axis while keeping the same range numbers. This could be prevented by adding additional stay constraints.

Bibliography

[ABB+92] Franz G. Amador, Deborah Berman, Alan Borning, Tony DeRose, Adam Finkelstein, Dorothy Neville, Norge, David Notkin, David Salesin, Mike Salisbury, Joe Sherman, Ying Sun, Daniel Weld, and Georges Winkenbach. Electronic "How Things Work" Articles. Technical Report 92-04-08, University of Washington, Department of Computer Science and Engineering, June 1992.

[Bar86] Paul Barth. An Object-Oriented Approach to Graphical Interfaces. *ACM Transactions on Graphics*, 5(2):142–172, April 1986.

[BMMW89] Alan Borning, Michael Maher, Amy Martindale, and Molly Wilson. Constraint Hierarchies and Logic Programming. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 149–164, Lisbon, June 1989.

[Bor81] Alan Borning. The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[Dui87] Robert Duisberg. Animation Using Temporal Constraints: An Overview of the Animus System. *Human-Computer Interaction*, 3(3):275–308, 1987.

[EL88] Danny Epstein and Wilf LaLonde. A Smalltalk Window System Based on Constraints. In *Proceedings of the 1988 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 83–94, San Diego, September 1988. ACM.

[EMB87] Raimund Ege, David Maier, and Alan Borning. The Filter Browser—Defining Interfaces Graphically. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 155–165, Paris, June 1987. Association Française pour la Cybernétique Économique et Technique.

[FBMB90] Bjorn Freeman-Benson, John Maloney, and Alan Borning. An Incremental Constraint Solver. *Communications of the ACM*, 33(1):54–63, January 1990.

[Hil90] Ralph D. Hill. A 2-D Graphics System for Multi-User Interactive Graphics Based on Objects and Constraints. In E. H. Blake and P. Wisskirchen, editors, *Advances in Object Oriented Graphics I*, pages 67–91. Springer-Verlag, Berlin, 1990.

[Hil92] Ralph Hill. Languages for the Construction of Multi-User Multi-Media Synchronous (MUMMS) Applications. In Brad Myers, editor, *Languages for Developing User Interfaces*, pages 125–143. Jones and Bartlett, Boston, 1992.

[Mal91] John Maloney. *Using Constraints for User Interface Construction.* PhD thesis, Department of Computer Science and Engineering, University of Washington, August 1991. Published as Department of Computer Science and Engineering Technical Report 91-08-12.

[MGD+90a] Brad A. Myers, Dario Guise, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Philippe Marchal, and Ed Pervin. Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment. *IEEE Computer*, 23(11):71–85, November 1990.

[MGD+90b] Brad A. Myers, Dario Guise, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Philippe Marchal, Ed Pervin, Andrew Mickish, and John A. Kolojejchick. The Garnet Toolkit Reference Manuals: Support for Highly-Interactive Graphical User Interfaces in Lisp. Technical Report CMU-CS-90-117, Computer Science Dept, Carnegie Mellon University, March 1990.

[MSB90] John Alan McDonald, Werner Stuetzle, and Andreas Buja. Painting Multiple Views of Complex Objects. In *Proceedings of the 1990 ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications and the European Conference on Object-Oriented Programming*, pages 245–257, Ottawa, Canada, October 1990.

[Mye87] Brad A. Myers. Creating Dynamic Interaction Techniques by Demonstration. In *CHI+GI 1987 Conference Proceedings*, pages 271–278, April 1987.

[Mye90] Brad A. Myers. A new model for handling input. *ACM Transactions on Information Systems*, 8(3):289–320, July 1990.

[Ols90] Dan R. Olsen, Jr. Creating Interactive Techniques by Symbolically Solving Geometric Constraints. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 102–107, Snowbird, Utah, October 1990. ACM SIGGRAPH and SIGCHI.

[San92] Michael Sannella. The SkyBlue Constraint Solver. Technical Report 92-07-02, Department of Computer Science and Engineering, University of Washington, October 1992.

[VZMGS91] Brad Vander Zanden, Brad Myers, Dario Guise, and Pedro Szekely. The Importance of Pointer Variables in Constraint Models. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 155–164, Hilton Head, South Carolina, November 1991.