Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

ThingLab II, Version 2

John Maloney
May 23, 1990

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195
USA

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

0. Disclaimer

ThingLab II is a research prototype and, as such, it's goal is to demonstrate concepts rather than to provide the world with a polished user interface construction tool. Thus, while we have tried to make it usable, it still has obvious blemishes and missing features and probably numerous bugs. We hope that you will see in ThingLab II the potential usefulness of constraints rather than the limitations of a particular system.

This manual was, of necessity, written in haste. We hope that this, too, will be forgiven. The alternative was to provide no manual at all!

1. Introduction

ThingLab II supports the exploration of constraint-based user interfaces. It consists of a set of classes that define constraints and constrainable objects called things. It also includes an incremental constraint satisfier, a module compiler, a construction-set style user interface, various tools, and an extensible set of primitive user interface building blocks.

ThingLab II uses the dataflow constraint model. In this model, a constraint is a collection of functions that use some subset of the constrained variables as inputs and compute the remainder as outputs. Each of these functions, called constraint methods (or methods, for short), can be executed to enforce the relationship represented by the constraint. For example, the constraint:

$$a = b + c$$

has three constraint methods:

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

```
a := b + c
b := a - c
c  := a - b.
```

Dataflow constraints can be used over a wide range of data types. For example, ThingLab II includes constraints that operate on numbers, bitmaps, strings, and lists. Dataflow constraints can also be executed

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

efficiently. However, they are not as powerful as some other kinds of constraints. For example, they cannot be used for linear programming, linear algebra, or scheduling problems. Furthermore, the particular dataflow constraint solver in ThingLab II cannot handle inequality constraints such as "x < 10" and is not guaranteed to find a solution if the constraint graph contains cycles. These limitations, along with the decision to use dataflow constraints in the first place, represent deliberate engineering choices. We believe that dataflow constraints, even with our restrictions, are sufficiently powerful for most user interface applications and the restrictions permit them to be implemented extremely efficiently. However, it is important to keep its limitations in mind to avoid asking ThingLab II to solve problems that it was not designed to handle.

The remainder of this manual presents a tutorial example, expands upon some of the basic concepts of ThingLab II, and briefly describes how to operate some of the tools. An appendix describes how to file ThingLab II into a Smalltalk-80 image.

2. Getting Started

To get started, invoke the "ThingLabII Parts Bin" item in the background menu. This gives you a view (i.e. a window) on the root of the parts bin hierarchy. There is initially only one parts bin, "All Parts," containing all the primitive Things. The "All Parts" bin will be updated as you work to contain all new Things you create. Open "All Parts" by selecting it and using the "open" menu item or by double-clicking. You should see a bunch of named icons like PointThing, LineThing, Sum, and MidPoint. These are primitive things.

Hint: Because the Mac mouse has only one button you can get the middle-button (yellow) menu by pressing the red button over the

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

title area of a ThingLabII window. The right-button (blue) menu is also available in the gray area around the title. This shortcut is not very useful on machines with three button mice!

Now create a new, empty thing by invoking the "new thing" menu item in one of the parts bin windows. Note that the new thing is given a unique name such as "Thing1" and an icon for it appears in "All Parts." The new thing's name and default icon may both be changed, if desired,

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

by selecting the thing's icon in "All Parts" and invoking the appropriate operation from the menu.

Components are added to the new thing by dragging them from a parts bin into the thing construction view. There is a modal dialogue involved when inserting new components. After you've dragged a group of components into the target thing and released the mouse button, the system expects you to specify where to place the parts by clicking the mouse once for each part. Add HLine and VLine things to the new thing and then pick up the endpoint of one of the lines with the mouse and move it around. You will notice a heavy black square appear when you move the point over any other point. This indicates that the points may be glued together ("merged"). If you release the mouse at this point, the merge will be done. You should now be able to construct simple polygons and rectangles (using LineThings, HLines, and VLines). Try it.

3. Concepts

Things

The primitive elements of the ThingLabII constraint programming system are variables and constraints. The unit of encapsulation used to assemble these elements into higher-level objects is the thing. A thing has a collection of parts, where each part is either a primitive variable or another thing. For example, a Node thing is composed of a primitive variable named ‘value’ and a Point thing named ‘location’. This Point thing is in turn composed of two primitive variables, named ‘x’ and ‘y’. In addition to its parts, a thing may also have a set of constraints that define relationships among its parts and subparts. For example, a HLine thing has such a constraint stating that the y values of its two endpoint Point things should be equal.

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

Cloning

New instances of things are created by copying an existing instance called the prototype. When a thing is copied, its structure of parts and subparts is copied all the way down to the leaves and its constraints are copied and installed in the new thing. The copying process is called cloning. A new kind of thing is created by starting with a new, empty thing, copying previously created things into it, and connecting these

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

things together with constraints. The resulting object is then the prototype for the new type of thing.

Merging

Identity relationships between parts of a thing may be established via merges. A merge, which may be thought of as a special sort of constraint, equates two subpart trees so that they become a single, shared part. After two subparts are merged, all constraints from the original subparts are applied to the new shared part. For example, the endpoints of two Line things might be merged. If the merged point is then dragged, both lines will be affected. Furthermore, if one of the original Line things was constrained to be horizontal and the other was constrained to be vertical, the merged point will now be governed by both constraints.

The Construction Kit Metaphor

The construction kit metaphor for thing construction has proven to be a powerful mechanism for packaging and reusing constraint “programs.” For example, a Quadrilateral may be constructed from four Line things. The Quadrilateral may be turned into a Rectangle by adding two horizontal and two vertical constraints. A center may be added by stretching a MidPoint thing across the diagonal. Finally, the centers and corners of several instances of the resulting CenterRectangle may be combined with additional vertical and horizontal constraints to produce a set of aligned boxes for a diagram or a paned-window layout. Note that many of the intermediate stages in this construction — Quadrilaterals, Rectangles, and CenterRectangles — are re-usable objects in their own right.

Symbolic Strengths

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

Constraint strengths are specified using Symbol objects such as #required. These are converted into Strength objects in various data structures. Class Strength keeps a table in a class variable that maps symbolic names to their indices in the table, which are used to order the set of Strengths. It is possible to insert new symbolic strengths into this table by modifying and then invoking Strength's class initialization method.

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

Constraints

Constraints may be defined either by using an equation:

```
	Constraint
		symbols: #(a b c)
		equation: 'a = (b + c)'
```

or by explicitly listing its constraint methods:

```
	Constraint
		symbols: #(a b c)
		methodStrings: #(
			'a := b + c'
			'b := a - c'
			'c := a - b')
```

In this example, the resulting constraints would be identical. The first form is preferred as it is compact and easy to read. There are times, however, when the equation translator cannot find an inverse function (such as when operating on bitmaps) or when one wishes to make a one-directional constraint. In these situations one must resort to the second form.

Note the parenthesis in the equation string in the first form. These are necessary so that the top-level expression passed to the equation translator is the "=" message send. If the parenthesis were eliminated the top level expression would be the "+" message send, since Smalltalk is evaluated left to right, and the equation translator would complain. (This is a blemish; it would be easy to make the equation translator figure this out for itself.)

Binding Constraints to Variables

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

Constraints are bound to their constrained objects using Reference objects. A Reference is a rooted symbolic path to a part or subpart of a thing. The "->" message can be sent to any thing to create a reference to one of its parts. For example: "myThing->#line1.p1.x". Note that the sequence of subpart names, "line1," "p1," and "x," are represented as a

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

single Symbol object. The symbol is broken into components at the period characters when the Reference is created. In some contexts, the root thing is implied, as in:

```
    aThing require: #node.location.x equals: #box.center.x
```

A previously created, unbound constraint may be bound to a set of variable references using the message "bind:strength:", as in:

```
    midPointConstraint
        bind: (Array
            with: self->#topLeft.x
            with: self->#center.x
            with: self->#bottomRight.x)
        strength: #required
```

Note that the strength must also be specified when a constraint is bound. Often, constraints are bound when they are created, as in:

```
    Constraint
        symbols: #(p1 midpoint p2)
        equation: '(p1 + p2) // 2 = midpoint'
        bind: (Array
            with: mpThing->#p1.y
            with: mpThing->#midpoint.y
            with: mpThing->#p2.y)
        strength: #stronglyPreferred
```

There are a number of shorthand forms for constructing and adding constraints to a thing, such as this example from the MidPoint primitive thing:

```
    mpThing
        stronglyPrefer: '(p1 + p2) // 2 = midpoint'
        where: #((p1 p1.y) (midpoint midpoint.y) (p2 p2.y))
```

Note that the strength is encoded in message selector and that the symbolic variable names and their paths (relative to root "mpThing") are compactly

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

specified in the "where:" clause. Many other shorthand forms can be found be browsing the protocols for Thing.

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

Adding and Removing Constraints

Confusing as it is likely to be, there are two senses in which constraints are added and removed. First, they are added to and removed from the constraint graph. A constraint does nothing, even after it has been bound to its variables, until it is added to the constraint graph with the message:

```
aConstraint addConstraint
```

It can be deactivated again by sending it the message:

```
aConstraint removeConstraint
```

The other sense in which constraints are added and removed has to do with them being owned by a thing. Any thing may own a set of constraints, and those constraints are cloned when the thing is cloned. Constraints attached to parts of the thing but not owned by it, such as mouse constraints, are not cloned with the thing. Constraints may be added to and removed from things using the following messages:

```
aThing addConstraint: mpConstraint
aThing removeConstraint: mpConstraint
```

Adding a constraint to a thing also adds it to the constraint graph as a side effect. (This sounds more confusing than it really is.)

Planning

One of the strengths of ThingLab II is the performance of its incremental constraint satisfaction planner. Constraint satisfaction is cheap enough that one may add and remove constraints dynamically and, in fact, the ThingLabII user interface does exactly this as the user interacts with the system. The incremental planner maintains a data flow graph among the constraints as constraints are added and removed. The dataflow graph represents a locally-predicate-better solution to the current set of constraints. (Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will necessarily be

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

satisfied. For details on how and why the constraint satisfier works, refer to the CACM article.)

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

The dataflow graph can be reduced to a linear list of constraint methods called a plan. The methods of the plan are executed in order to compute a solution to the current set of constraints. In the original ThingLab, this list of methods would have been compiled into a Smalltalk method. Although the code generated was quite fast, the compilation process itself was expensive and had to be repeated each time the constraint graph was modified.

Module Compilation

Unlike the original ThingLab, ThingLabII does not normally compile plans into Smalltalk methods. However, when a given thing has been developed to the point of stability and would be useful as a building block for constructing other things, it may be compiled into a module. A module behaves externally like the thing from which it was compiled, but with better performance. All possible plans for satisfying the module's internal constraints are pre-computed, optimized, and compiled into Smalltalk methods. The module's planning behavior is similarly pre-computed so that it appears to the planner to have only a single (albeit complex) internal constraint.

Constructing Things

New kinds of things may be constructed either by using the direct-manipulation interface or by writing a program to do the construction. A program can do anything that can be done using the direct-manipulation interface plus various things which are awkward to do via direct manipulation, such as using nested loops to interconnect an array of components with regular structure (e.g. laying out a chess board). The direct-manipulation interface is also not a good vehicle for adding constraints that are not built into some graphical object, since there is no way to view and manipulate these “invisible” constraints. The demo classes are

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

examples of how to construct things using programs.

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

Adding Primitive Things

A new primitive thing is added by creating a new subclass of PrimitiveThing. Three initialization methods are used to define the structure of the new thing:

> initializeStructure,
> initializeConstraints, and
> initializeValues.

The parts of the thing are defined in initializeStructure. Any part that holds a thing must be initialized here. Instance variables not initialized to be things are assumed to hold non-thing values which do not need to be recursively copied during cloning. Constraints are defined in initializeConstraints. Finally, initial values are declared in initializeValues. Any of these initialization methods may be omitted if desired; the default behavior is to do nothing.

By convention, all primitive things have a class initialization method that initializes their icon bitmap for the parts bin and their explanation string. The class initialization method must do "self initializePrimitive" before doing anything else. If the class initialization method is omitted, a default method provides a generic icon bitmap and explanation string.

If the new primitive is to have custom appearance, it must supply a display method and various other glyph protocol methods. Likewise, if it to have custom mouse or keyboard input behavior, it must supply methods to support this behavior. Unfortunately, although it is not difficult, it is beyond the scope of the introductory manual to describe in detail how to add such behavior. The best way to learn how to do it is to read the comments in class Thing and study the supplied primitive things.

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

Time and State

ThingLabII captures the notion of state changes in time via a mechanism called history variables. A history variable stores not only its current value, but some fixed number of past values as well. Constraints may refer to the values of previous states but may not alter

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

them. This forces time to always move forward and allows the implementation to truncate histories at some reasonable limit. A system clock advances the histories of all history variables in a given thing as a single atomic operation.

This simple model allows one to elegantly express time-dependent behavior. For example, a time-varying variable can be integrated by using a Sum constraint to add its current value to the previous value of the variable containing the integral. Similarly, the discrete-time derivative of a time-dependent variable can be computed by taking the difference between successive states. More relevant to user interface construction, one can build finite state machines for processing user inputs or producing simple animations. As another example, the browser demo uses the history mechanism to pre-select the most recently selected message category and message pane selections, if possible, when the class pane selection is changed.

Constraints and Imperative Code

A user interface must inform the application program of user actions. Similarly, the application program will take actions, autonomously or in response to user actions, that will effect the graphical objects visible in the user interface.

It is best to think of the world in two parts: the world of constraints and constrained variables (the constraint world) and the less disciplined world of normal Smalltalk programs (the imperative world). The imperative world is in control but interacts with the constraint world by:

1. instantiating a set of variables,
2. adding and removing constraints on those variable,
3. examining the values of constrained variables,

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

4. changing the values of constrained variables, and
5. invoking the constraint satisfaction machinery.

Interacting with the graphical objects of a user interface occurs through a Smalltalk View-Controller pair that knows about the constraint world. For example, moving a point causes mouse constraints to be added to the point, all constraints to be repeatedly satisfied as the

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

mouse moves, and the mouse constraints to be removed at the end of the interaction. The ThingLab II user interface framework has hooks that allows custom widgets, such as sliders and buttons, to be handled in a similar manner. Thus, the ThingLab II UI is simply a special imperative program that interacts with the constraint world.

The application program may also interact with the constraint world. The rules for this are simple. Any variable may be examined at any time. However, the application must be only change the values of variables in a way that allows constraints on the variable to be kept satisfied as well as possible. One way to do this is with the set:to:strength: message. For example, one could write:

```
        p set: #x to: 15 strength: #preferred
```

This statement can be read: “I would prefer that the x part of point p be 15 now.” It is implemented by adding an edit constraint with a strength of preferred to the x part of p and invoking the planner. If the planner finds a way to satisfy the edit constraint, then the value of x is changed and the plan is executed; otherwise, the value of x is left unchanged. Finally, the edit constraint is removed.

Because the set:to:strength: mechanism invokes the planner twice (once to add and once to remove the edit constraint) it is not efficient if a sequence of changes must be made to the same variable, such as during an animation sequence or a drag interaction. In such cases, the client program must add edit constraints for all variables that will be changed, extract a plan, and invoke the plan after every set of variable change. This is how the dragging is implemented in the ThingLab II UI.

The mechanisms just described are clearly not as easy to use as one would like and should be thought of as work in progress. Many of the issues raised will be addressed in Bjorn Freeman-Benson’s

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

thesis on constraint-imperative programming.

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

4. The User Interface

Gestures

The system recognizes several different gestures made with the mouse, namely: click, double-click, and drag. All gestures are made with the red button of the mouse and possibly the shift key, and are context sensitive.

A drag occurs when the mouse button is held down longer than about a quarter second. A special kind of drag, called a "sweep", is made by moving the mouse down and right quickly during the initial mouse press. This is used for the "area select" operation.

Selecting

The system maintains a list of selected objects which are used as arguments to menu commands or input actions. Sweeping is used to select multiple objects to operated on, with feedback given by drawing a temporary rectangle around the area to be selected. Shift-clicking or shift-sweeping toggles the selection of the designated objects. Clicking over an object makes it the only thing selected. Clicking over the background clears the entire selection. Double clicking over an object "opens" it. (This is an example of gesture whose meaning varies with context. In the parts bin, opening an parts bin brings up a new window containing its contents while opening a thing brings up an editor on the thing. In a thing editor window, double-clicking on a part brings up an inspector window on the part.) Double- clicking over the background does a select-all operation. Non-sweep drag gestures over the background are used for the "scroll" operation (reflected by a little hand cursor). All operations except moving and selecting objects can also be invoked via the menu.

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

Parts Bins

Primitive and previously constructed things are kept in a set of hierarchically nested, iconic parts bins. The root of the hierarchy is called “Top Bin”. A special bin called “All Parts” contains icons for all primitive and constructed things in the system and is the only parts bin from which a thing may be deleted from the system. The user may construct additional parts bins to organize his set of things as desired.

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

The icon for a given thing may reside in any number of parts bins, allowing Things to be cross filed in a number of bins.

Editing Things

Most Things may be moved by dragging them with the mouse. Several objects may be selected and dragged together. Such interactions are accomplished by adding mouse constraints to the location parts of the objects to be moved. If any of these locations is fixed by a constraint of higher strength, that object cannot be moved. It is possible to increase the strength of the mouse constraints using the ThingLab II Control Panel. This is useful when one wishes to move the PointAnchor.

Editing non-graphical values not quite so obvious. You specify numbers by selecting one or more NumberPrinter or NumberDisplayer things and typing digits. Constraints are resatisfied as you type. Backspace deletes the last digit, the minus sign changes the sign, and the period may be used to input floating point quantities. Strings may by typed into TextThings in a similar manner.

To draw bits into a FormDisplayer, use the yellow button (the red button is used to select and move it). You may also bring up a fat-bits editor on the form by using shift-yellow button.

As a last resort, you can edit the value of an object using a Smalltalk inspector. This goes outside the normal constraint world, however, so constraint satisfaction will not occur until you do something else to trigger it. You can open an inspector by selecting a single object and invoking the "inspect" menu item or by double-clicking over the object. If there is no selection, invoking the "inspect" menu item will open an inspector on the top-level thing.

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

Using Custom Constraints

A custom constraint allows the user to define a new constraint directly from the direct manipulation interface. There are two-, three-, and four-variable custom constraints. Initially, a custom constraint does not constraint its variables in any way. Shift-clicking on the custom constraint brings up a Constraint Definer view. The user types in the methods of the constraint separated by blank lines. A method consists of

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

one or more assignment statements. The constrained variables listed at the top of the view should be used in these statements. It is also allowable to reference global variables such as "Transcript". When done, the use invokes "accept" from the menu and the constraint is installed. Since the constraint is usually executed as soon as it is installed, the user should be sure that the variables have reasonable values (i.e. not nil) before installing the constraint. To remove a custom constraint, delete all the methods and "accept" again. This will return it to its initial, unconstraining, state.

The Module Compiler

The Module Compiler is invoked with the "make module" menu command in a thing construction view. This changes the view to one used to specify the parts that should be externally visible after the module is compiled. In this view, parts to be externally visible are displayed normally and all other parts are shown in gray. External parts may be toggled by shift-clicking on them. When the external parts have be designated, the "compile" menu command is invoked. The compiler presents a picture giving feedback as it goes through the compilation process. Then, the view is changed back to a thing construction view, but now it shows the compiled module instead of the original thing. You may interact with this module to verify that it behaves correctly. The "view source" and "view module" menu commands can be used to switch between the source thing and the module compiled from it.

Warning: The Module Compiler has not been tested extensively and almost certainly has lingering bugs.

The Debugger

The debugger allows the user to study the constraints of a thing through a graphic display of the underlying constraint/dataflow

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

graph. It is invoked with the "debugger" menu command in the thing construction view. The nodes in this constraint graph represent variables and the arcs represent constraints. Variables are labeled with their path and constraints are labeled with abbreviations representing their strengths.

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

The graph is not presented all at once but rather in independent subgraphs called partitions. By definition, there are no inter-partition constraints and thus each partition behaves independently of all other partitions. The number of partitions is displayed in the upper left corner of the debugger view along with two arrows. Clicking on the arrows with the mouse cycles forward and backward through the partitions.

Constraint arcs are labeled with arrow heads to show which constraint method was selected by the planner for each constraint. If the constraint is not currently satisfied, its arc is displayed in gray. The debugger initially shows the dataflow graph for the current solution. There may be multiple, equally good solutions. These may be cycled through using the arrows. The first time one of the arrows is pressed, all solutions are computed (which may take a while) and then the number of solutions is displayed. Only the arrowheads and gray/non-gray status of the constraint arcs changes when the displayed solution changes.

Constraints and variables and be moved with the mouse to create a pleasing and readable layout. In addition, several menu commands help achieve a good layout. The "center constraints" command centers constraint labels between the constrained variables. The tails of constraints with only one variable, such as stay, mouse, and edit constraints, are made to point away from the center of the constraint graph in an attempt to keep them out of the way. The "layout" command invokes a more powerful but slower graph layout algorithm. The graph is redisplayed as the algorithm executes and the user may press and hold the mouse button to force the layout algorithm to terminate early.

Warning: The layout algorithm becomes very slow for large graphs. However, even when nicely laid out, large graphs are difficult to understand so perhaps some sort of modularity mechanism is

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

needed to limit the amount of detail presented to the user.

5. Release Notes

This version of ThingLab II runs only on version 2.3 of Smalltalk-80 from ParcPlace Systems. It should, however, port fairly easily to other standard Smalltalk-80 systems. For example, we ported an earlier version to Tektronix Smalltalk in half a day. Unfortunately, it would be

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

difficult to port the system to Digitalk Smalltalk because in their system the compiler classes are not available to the end-user and the equation translator and module compiler construct and manipulate Smalltalk parse trees.

This is the second release of ThingLab II. The first release was given out to only a few other research laboratories. This version fixes a number of the bugs and limitations of the first release and improves performance considerably.

Although we haven't the resources to maintain ThingLab II, we are definitely interested in your experiences with the system, including bug reports and suggestions. Comments should be addressed to:

John Maloney (jmaloney@june.cs.washington.edu)

who may also be reached at:

Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195
USA

6. References

The following are additional references for ThingLab II. Pointers into the general literature on constraint programming may be found in each paper, although the bibliography of the CACM paper is the most comprehensive.

Maloney, J., Borning, A., and Freeman-Benson, B. "Constraint Technology for User Interface Construction in ThingLab II" In OOPSLA '89 Proceedings, pp. 381-388.

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

Freeman-Benson, B. "A Module Mechanism for Constraints in Smalltalk" In OOPSLA '89 Proceedings, pp. 389-396.

Freeman-Benson, B., Maloney, J., Borning, A. "The DeltaBlue Algorithm: An Incremental Constraint Hierarchy Solver" CACM 33:1 (January 1989), pp. 54-63. Available in expanded form as Department of

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

Computer Science and Engineering Technical Report 89-08-06, University of Washington, Seattle, WA, 98195.

Appendix: Installing ThingLabII

The following three files should be filed in, in order:

    ThingLabII.v2.st
    Things.v2.st
    Demos.v2.st

This will take a while, perhaps as much as an hour on some platforms. There is also an optional .form file to be placed in the same directory as your image:

    ThingLabII.form

If available, this file is used to display a humorous picture when the image is started.

Acknowledgements

The first version of ThingLab II was implemented from scratch by Bjorn Freeman-Benson and John Maloney in about four months. It was further developed and improved over the following year by John Maloney. Alan Borning acted as high-level consultant and mentor.

ThingLab II incorporates many of the ideas from Alan Borning's earlier constraint programming system, ThingLab, including constraint hierarchies, merging, and the construction-set metaphor. Spiritual ancestors include Gosling's thesis and Sutherland's ground-breaking SketchPad system.

Optimal, that is, according to the locally-predicate-better measure of the goodness of a solution to a given constraint hierarchy. Since there may be conflicts between constraints and some constraints are stronger than others, not all the constraints will be satisfied. For the specifics, see [a bunch of papers].
\

Apple Computer provided two Macintosh II computers and a one year fellowship for John Maloney. IBM also supported John Maloney for one year. An NSF fellowship supported Bjorn Freeman-Benson. Additional funding was provided by NSF Grant No. IRI-8803294 and the Washington Technology Center. We are grateful to all for their support.