

Using Events for constraint solving

Annick Fron

Smalltalk has evolved and the good old MVC scheme has now grown up into events. Events are implemented in Visual Smalltalk and Visual Age, but the code will be presented here in the Visual Smalltalk environment.

The MVC implementation in Smalltalk relies on a couple of changed/update messages sent back and forth between dependents.

Figure 1 shows how the dependency mechanism is triggered: Each time object A is modified and calls method "changed", or one of its variants, all of its dependents are informed and execute an "update" method.

A more sophisticated scheme allows an aspect to be passed as an argument to an update, in order to refine the monolithic link between dependent variables. The problem is that the update method has to decode the argument to decide on its behavior. For example,

```
anObject changed: #color
anotherObject >>update: anArgument
    anArgument = #color ifTrue: [...]
    anArgument = #size ifTrue: ...
```

Event programming can be seen as a refinement of the *changed/update* pairs. Namely, instead of maintaining a list of dependents, the system maintains a dictionary with events as entries. This allows for a much more efficient and finely tuned scheme for maintaining dependencies.

Event-based programming has been very successful for building graphical interfaces and for visual programming. In Visual Age and Visual Smalltalk, events are the backbone of links between objects in order to build a visual application.

Events are also popular at the operating and windowing system levels, but this is not the topic here.

CONSTRAINT-SOLVING TECHNIQUES

Constraint-solving techniques allow the user to tackle such combinatorial problems as scheduling a meeting or dispatching resources according to some criteria.

One popular technique, called "Finite domain tech-

niques," is used when variables can only take a finite number of values. For instance, scheduling a project when the time unit is a day can be modeled using this technique.

The idea is to represent the variable not as bounded to a value, but with a range of potential values called its domain. The solver will then ensure consistency between these domains through the constraints.

For instance, if two variables are constrained to be equal, their domains should be the same. Hence, each time one domain is modified the other one should be as well.

The easiest way to represent domains is through intervals. One easy way to detect interval modification is through its bounds. On intervals, it is thus possible to define two modification events, on each bound. We will call them: #min and #max.

In Figure 2, two constrained variables and one constraint are shown. When the domain of *x* is modified by some other constraints or external event, it informs all connected constraints, in this case here the equality constraint. This constraint has only one connected variable, *y*. Because it is equal, it must tell *y* to modify its lower bound as well, thus triggering a #min event for *y*. *y* in turn will inform its own constraints.

Consistency on finite domain has been proven to be efficient for solving constraints on integers, which is usually exponentially complex. The only restriction to get a fixed-point solution is to always shrink the domain; never

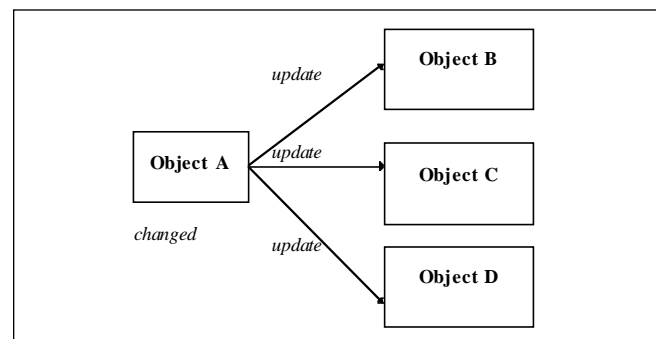


Figure 1. A changed message in object A triggers update messages on all its dependents.

increase it. The triggering order of events on domains has been proven not to affect the result. Yet, this algorithm is not complete and needs an enumeration phase in order to find all of the solutions. This will be omitted here.

A SMALLTALK IMPLEMENTATION OF CONSTRAINTS

The Smalltalk implementation of a constrained variable is very simple, and events will help a lot in the job. The first question to ask is: "Do the constraints represent objects, or are they included into other objects?" Having constraints as true objects is very helpful. Because it allows the implementation of a constraint hierarchy to refine new constraints, it provides a handle to dynamically inhibit or activate a constraint.

Here, we take a simple assumption that constrained variables need not be any Smalltalk object; they can derive from a specific root under object. Therefore, we get two object hierarchy roots, one for constraints, one for variables.

THE VARIABLES

Variables are defined by their domain and by the events they are able to respond to. Here we get only finite domain variables (the model can be extended to other kinds of variables, but for reasons of simplicity, it will be omitted here).

In Visual Smalltalk, it is possible to define any semantic event at the class level, through the method `constructEventsTriggered`. The second step is to couple every domain modification (instance variable accessor) with an event using the `triggerEvent: message`. This should be compared to adding changed to instance variable accessors displayed on a view.

The operator `*` and `=@` are only syntactic sugar for the final example. `=@` means equals in the constraint sense, which should not be confused with variable assignation (variables that have no value yet).

For esthetic reasons, variables get a name to be printed.

Object

```
subclass: #ConstrainedVariable
instanceVariableNames: 'name domain'
classVariableNames: ''
poolDictionaries: ''
```

!ConstrainedVariable class methods!

constructEventsTriggered

```
"Private - answer the set of events that instances of the
receiver can trigger."
^#(#min #max) asSet
```

from: aMin to: aMax

```
^super new from: aMin to: aMax
```

!ConstrainedVariable methods!

```
* y
```

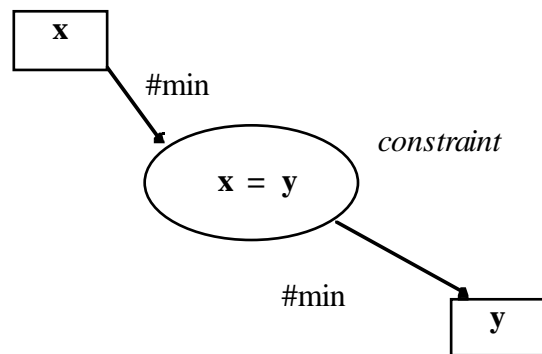


Figure 2. An event on a variable will propagate to all related constraints. The constraints in turn will trigger new events on the connected variables.

```
| c |
c := ActTimes x: self y: y.
^c result "a constrained variable"

+ y
| c |
c := ActAdd x: self y: y.
^c result "a constrained variable"

@= y
| c |
c := ActEquals x: self y: y.
^c result " a constrained variable"

domain
^domain!

domain: anObject
domain := anObject!

max
^domain last!

max: aValue
domain newMax: aValue.
self triggerEvent: #max!

min
^domain first!

min: aValue
domain newMin: aValue.
self triggerEvent: #min!

printOn: aStream
aStream nextPutAll: name, ' ', domain printString!

from: aMin to: aMax
domain := Interval from: aMin to: aMax.
```

THE CONSTRAINTS

Here we need two types of constraints: constraints on comparators, which will have two arguments, `x` and `y`, as pointers to the variables they involve; and constraints on operators when we need to introduce a new constrained

variable, the result, which is also stored as an instance variable and called r.

When a constraint is created in the program, the method post will define the event dispatch mechanism according to the constraint semantic. It uses the when:send:to: message that links events to actions on several objects.

The following table shows the links between events and domain updating for the equality constraint $x @= y$.

Description	Event	Update
x minimum increased	#min on x	y domain takes min(x) as new lower bound
x maximum decreased	#max on x	y domain takes max(x) as new upper bound

Symmetrically, we can define the same for y. Methods have to be defined in the constraint class to compute the updates, such as xmin or xmax. The constraint is called ActEquals. ActEquals also needs an init method in order to ensure domain consistency prior to any computation.

In the example, the two operators introduced are ActAdd (addition) and ActTimes (multiplication by a constant factor). These classes inherit from ActConstraint, an abstract class which does not have any behavior here.

ActConstraint subclass: #ActEquals

```
instanceVariableNames: 'x y'
classVariableNames: ''
poolDictionaries: ''!
```

!ActEquals class methods !

```
x: var1 y: var2
^super new x: var1 y: var2.
```

!ActEquals methods !

```
x: var1 y: var2
x := var1 . y := var2.
self post; init.
```

init

```
| m s |
m := x min max: y min.
s := x max min: y max.
x min: m; max: s.
y min: m; max: s
```

post

```
x when: #min send: #xmin to: self.
x when: #max send: #xmax to: self.
y when: #min send: #ymin to: self.
y when: #max send: #ymax to: self
```

xmax

```
y max: x max
```

xmin

```
y min: x min
```

ymax

```
x max: y max
```

ymin

```
x min: y min
```

=====

ActConstraint subclass: #ActAdd

```
instanceVariableNames: 'x y r'
classVariableNames: ''
poolDictionaries: '' !
```

!ActAdd class methods !

```
x: var1 y: var2
^super new x: var1 y: var2
```

!ActAdd methods !

x: aVar1 y: aVar2

```
x := aVar1
y := aVar2.
r := ConstrainedVariable from: (x min + y min) to: (x max + y max).
self post
```

post

```
x when: #min send: #xmin to: self.
x when: #max send: #xmax to: self.
y when: #min send: #ymin to: self.
y when: #max send: #ymax to: self.
r when: #min send: #rmin to: self.
r when: #max send: #rmax to: self.
```

rmax

```
x max: (r max - y min).
y max: (r max - x min).!
```

rmin

```
x min: r min - y max.
y min: r min - x max!
```

xmax

```
r max: x max + y max.
y min: r min - x max
```

xmin

```
y max: r max - x min.
r min: y min + x min
```

ymax

```
r max: x max + y max.
x min: r min - y max
```

ymin

```
x max: r max - y min.
```

CONSTRAINT SOLVING

r min: x min + y min

ifTrue: [r max: y * x max]

ifFalse: [r min: y * x min]

=====

ActConstraint subclass: #ActTimes

instanceVariableNames: 'x y r'

classVariableNames: "

poolDictionaries: " !

!ActTimes class methods ! !

x: var1 y: anInteger

^super new x: var1 y: anInteger

!ActTimes methods !

x: aVar1 y: anInteger

x := var1. y := anInteger.

anInteger >= 0

ifTrue: [r := ConstrainedVariable from: anInteger * x
min to: anInteger * x max]

ifFalse: [r := ConstrainedVariable from: anInteger *
x max to: anInteger * x min].

self post.

post

x when: #min send: #xmin to: self.

x when: #max send: #xmax to: self.

r when: #min send: #rmin to: self.

r when: #max send: #rmax to: self.

rmax

y >= 0

ifTrue: [x max: (r max / y) floor]ifFalse

: [x min: (r max / y) ceiling]

rmin

y >= 0

ifTrue: [x min: (r min / y) ceiling]

ifFalse: [x max: (r min / y) floor]

xmax

y >= 0

xmin

y >= 0

ifTrue: [r min: y * x min]

ifFalse: [r max: y * x min]

=====

EXAMPLE

The simple example is used to test the code and show how some partial solving can be achieved. It defines the domains of the variables, and sets the only constraint: $x + 3y + 4z = 2t + c$.

ConstrainedVariable class>>example

"ConstrainedVariable example"

| x y z t c |

x := ConstrainedVariable from: 0 to: 3. x name: 'x'.

y := ConstrainedVariable from: 0 to: 1. y name: 'y'.

z := ConstrainedVariable from: 2 to: 5. z name: 'z'.

t := ConstrainedVariable from: 0 to: 3. t name: 't'.

c := ConstrainedVariable on: #(5). c name: 'c'.

((x + (y * 3)) + (z * 4))@=((t * 2) + c).

x printOn: Transcript.


y printOn: Transcript.

z printOn: Transcript.

t printOn: Transcript.

Transcript cr.

COMMERCIAL IMPLEMENTATIONS

This article has given a brief insight into constraint solving techniques. These techniques have been commercially implemented in C++, and used on such industrial applications as train and plane scheduling. Smalltalk events allow a very elegant presentation of the consistency scheme. 

Annick Fron can be reached at 100342.3301@compuserve.com.