Implementing a propagator in Gecode/J

Mikael Z. Lagerkvist

This document gives a short introduction on how to write a basic propagator in Gecode/J.

Propagators in Gecode/J

All propagators in Gecode/J are subclasses of the class `Propagator` in the package `org.gecode`. In contrast to scripts, a propagator uses so called variable views. A view of a variable will expose methods for modifying the domain of the variable directly, for copying the view, and for subscribing/cancelling subscription to modifications of the view. A propagator in Gecode/J will subscribe to changes in variables by using a variant of the event-sets presented in the lectures called *propagtion conditions*.

For an example of domain-updates, the interface `IntView` in the package `org.gecode` conains the method `lq(JavaSpace home, int i)`, which updates the domain of variable in question to be less than `i` in space `home`. The methods for updating the domain of a view returns a `ModEvent` indicating the result of the tell. Your must *always* check the return value of these methods for failure. See below on how to signal failures. To get an IntView represting an IntVar, use the class `IntVarView`. To store an array of IntViews, use the class `ViewArray`.

In Gecode/J, the event-sets presented in the lectures are called propagation conditions. These conditions are used by a propagator when it subscribes to the views it is interested in. A propagation condition is a member of the `PropCond` enumeration in the `org.gecode` package. This enum has three members, `PC_INT_VAL`, `PC_INT_BND`, and `PC_INT_DOM`. The first is used when a propagator is only interested in value assignments (the $\text{fix}(v)$ event from the lectures). The second is used when a propagator is interested in changes to the bounds of its views (a $\min(v)$ event or a $\max(v)$ event has occured). The last

propagation condition corresponds to the any(v) event, which means that something has changed.

Implementation of propagators

There are some convenience classes available in the package `org.gecode` for subclassing that take care of handling the views (copying the variables, subscribing to them, as well as cancelling subscriptions). These are `BinaryPropagator` (for propagators over two views), `NaryPropagator` (for propagators over a ViewArray), and `NaryOnePropagator` (for propagators over a ViewArray and an additional single view). For your propagator (assuming that you subclass one of the convenience-classes), you will need to implement the following methods and constructors:

- A constructor for creating the propagator.
- A copy-constructor (taking a home-space, a Boolean share argument, and the propagator to copy).
- A `cost()` method that returns the cost of running this propagator. The possible return values are given by the `PropCost` enumeration in the `org.gecode` package.
- A `propagate` method (taking a home-space). This method should contain the filtering algorithm for your propagator. The method should return an element of the `ExecStatus` enumeration in the `org.gecode` package.
- A static `post(...)` method. This method is the method called when you want to post a constraint. It should construct a new Propagator object, and call the `addPropagator(Space,Propagator,boolean)` method in the `Gecode` class.

You should make sure that your `propagate` method signals when/if it is at a fixpoint (by returning `ES_FIX`) and when it is subsumed (by returning `ES_SUBSUMED`) correctly. Also, make sure that you *always* check for failure when updating the domain of views, and return `ES_FAILED` when that happens. If none of the above applies, return the value `ES_NOFIX`.

Tips

For examples of how to write propagators, see the `QueensJavaPropagator` example distributed with Gecode/J. This example contains an implementation of the binary

not-equals constraint, as well as a simple implementation of the distinct-constraint (corresponding to the value-consistent distinct in Gecode).