Architectures for Cooperating Constraint Solvers

Alan Borning

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VPRI Memo M-2012-003

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

Architectures for Cooperating Constraint Solvers

Alan Borning

18 March 2012

1 Introduction

We want to have an extensible collection of constraint solvers that can interoperate. As we've discussed, this fits well with Problem Oriented Languages — a POL could have one or more associated constraint solvers. (The alternative of having just a fixed, built-in set of solvers seems too inflexible and doesn't lend itself to extensibility.)

The kernel could have a couple of built-in solvers, but these shouldn't have special status with respect to additional solvers in terms of performance or capabilities. So we need to figure out the API for a solver, how they interoperate, and how the right solver or solvers are selected to solve a problem.

Here are some features that may be desirable for the system of cooperating solvers:

- Support for soft as well as required constraints
- Incremental resolving (for example, when given a sequence of inputs resulting from moving the cursor)
- Compiling constraint satisfaction plans
- Support for finding multiple solutions (Worlds)
- Support for debugging and understanding. If the system can't find a solution, why not? What could you do to let it find one? If it found an unexpected solution, where did it come from? How could you nudge it to find something more reasonable?
- Support for temporal constraints, so that, for example, we can simulate springs and oscillation

2 Prior Work

There is a reasonable amount of prior work in this area. (This discussion is unfortunately biased toward the older work that I know better — particularly

VPRI Memo M-2012-003

if this note were to become part of an academic paper, this discussion should be updated with recent results.)

As with many things, Sketchpad [13] was one of the first, if not the first, system to provide multiple cooperating solvers. Sutherland's dissertation describes two solvers: relaxation (an iterative numerical method that minimizes the errors in satisfying the constraints), and the "one pass method" (propagation of degrees of freedom). I've been told that Sketchpad also used a third solver: propagation of known values. ThingLab [1] provided these three solvers as well. A limitation of both of these systems was that the set of solvers are built-in rather than being easily extended. Another limitation was that there wasn't a declarative semantics for constraints, in particular how constraints interact with state, so that if a new solver could be added, it would be hard to argue convincingly that it was giving the "correct" answer and interacting properly with existing solvers.

Nelson and Oppen [11], working in the context of program verification, presented a way to combine multiple decision procedures. There is now a considerable literature in this area of SMT solvers (Satisfiability Modulo Theories). The solvers communicate by shared variables, inferring equalities on those variables. This is powerful but has certain limitations. The usually-noted limitation is that the theories need to be convex for the decision procedure to be complete. So, for example, the theory $(\mathbb{Z}, +)$ and equality is convex, but $(\mathbb{Z}, +, \leq)$ is not. In other words, this method isn't complete (i.e., may not find the answer) for inequality constraints over the integers. Perhaps more importantly for the current project, the theory doesn't accomodate soft constraints.

The Constraint Logic Programming framework [8] also has cooperating decision procedures (constraint satisfiers), namely for tree constraints (pure Prolog, which just has equality constraints over terms from the Herbrand universe) and some other domain. For example, CLP($\mathcal{R}$) [9] combines tree constraints and constraints over the real numbers.

Closer to the Sketchpad/ThingLab lineage, Bjorn Freeman-Benson and I developed Ultraviolet [3], a constraint satisfaction algorithm that invokes different subsolvers for different parts of the constraint graph. In the version we implemented, the available solvers are:

- a local propagation solver for functional constraints (Blue – a non-incremental version of DeltaBlue [12])
- a local propagation solver for constraints on the reals, including inequalities (Indigo [2])
- a solver for simultaneous linear equations (Purple)
- an *incomplete* solver for simultaneous linear equations and inequalities (Deep Purple)

Ultraviolet supported required and soft constraints, and compiling plans for repeatedly solving collections of constraints.

How it worked: we can view the constraints and constrained variables as forming a bipartite graph [5]. Each variable and each constraint is represented by a node, with an edge from a constraint node to a variable node if the variable is constrained by the constraint. The constraint graph is acyclic if the bipartite graph is acyclic. Ultraviolet partitioned the constraint graph into cyclic and acyclic regions. One can view each region as itself a node in a larger graph; the overall graph is acyclic. Every constraint belongs to exactly one region, but a variable can be shared by more than one region (which is how the subsolvers communicate). Ultraviolet then found an appropriate solver for each region. The key to its operation is that the constraints are solved level by level rather than region by region: first solve the required constraints and propagate any information among the solvers, then solve the strongest of the soft constraints and propagate information, and so forth.

Ultraviolet had some nice features, but also some major limitations, which led me to abandon this line of work once we designed and implemented Cassowary. The main limitations were:

- It required solvers that could process soft constraints level by level. (Cassowary doesn't support this — Deep Purple did, but it is not a complete solver.)
- Compilation was slow (even though execution was fast). This made it unsuitable for situations where the set of constraints was changing rapidly, as in an interactive graphical application.

The DETAIL system by Hosobe et al. [6, 7] was quite similar to Ultraviolet, and supported hard and soft constraints, and also included a meta-solver that supports different subsolvers.

3 Constraint Theory: Soft Constraints and Read-only Annotations

At least for some kinds of problems (including using constraints for interactive graphics) I think it's important to have soft constraints, not just required ones. Among other things, soft constraints provide a clean, declarative semantics for expressing the desire that parts of a graphical object stay where they were unless there is some good reason for them to move. (As long as we are talking ancient history, the "frame problem" of McCarthy and Hayes [10] concerns the same issue.) Both Sketchpad and ThingLab supported this, but hard-wired into the implementation of the solver rather than being part of the declarative specification. The "constraint hierarchy" theory [4] provides one way of doing this — I still like it as a theory, although as we've discussed I'd change the terminology to call them "soft constraints" and "priorities" rather than a constraint hierarchy.

Another useful feature is read-only annotations. The intuition behind this is that a constraint should not be allowed to affect the choice of values for its read-only variables. For example, if we have a constraint that a point follow

the mouse, the constraint should be read-only on the mouse position (unless, of course, the mouse is equipped with a small computer-controlled motor). Another use is in constraints describing a change over time, where the constraint relates an old and a new state. Here, we will probably want to make the old state read-only, so that the future can't change the past. Read-only annotations for constraints date back to Sketchpad, but in both Sketchpad and ThingLab they were defined operationally, in terms of how the solvers function. More recently (although still long ago), we were able to give a declarative meaning for read-only annotations, independent of how solutions are found [4].

4 Proposed Architecture

As noted in the previous section, we should support soft constraints, at least in some parts of the system. But, at least as far as I can see, doing so in an architecture for cooperating solvers puts some difficult requirements on the subsolvers. In particular, they should be able to handle soft constraints priority by priority (as in Ultraviolet); and there may well be other strong requirements on the cooperating solvers if we want to have a comprehensive architecture.

So we might retreat to a very simple architecture that puts minimal requirements on the component solvers. But then the system might be too weak to solve very many interesting sets of constraints. Fortunately, I think there is a way here to have our cake and eat it too! The idea is to use multiple levels of cooperating solvers. At the top level, we use the simplest of architectures, which has a straightforward dataflow solution. Then, within each region, we can either use a single solver, or else have another layer of cooperating solvers (but perhaps using a more sophisticated architecture to join them).

As with Ultraviolet, let's view the constraints and constrained variables as a bipartite graph. For the top level, we partition the constraints into regions, based on the following requirements:

- each constraint belongs to exactly one region
- the different regions can share variables (but not constraints)
- for each variable shared by two or more regions, that variable must be read-only for all but one of the regions
- the graph of the different top-level regions must be acyclic
- it isn't possible to divide any region into two regions and still maintain the other requirements

Note that we can always partition the constraint graph this way. (In the limit, there is a single region.) The partitioning is unique.

Given this partitioning, and a way to solve the constraints in each region, it's easy to solve the constraints as a whole: it's just a dataflow problem. We find an order for solving the regions, so that region B is solved after region A if B is downstream from A in terms of the read-only variables.

Furthermore, at the top level we may be able to accommodate other kinds of Problem Oriented Languages, not just constraint languages — these other regions just need to have external variables so that they can be coupled (again, with the restriction that all but one use of each variable must be read-only).

Incidentally, I believe that the restricted information flow among regions at the top level is a subset of what is allowed in both SMT[1] and in Ultraviolet. But for information flow in SMT and Ultraviolet, neither is a subset of the other: SMT allows equalities between otherwise unbound variables, and Ultraviolet can propagage interval constraints on variables (for example, that $x \in [1, 5]$). This proposal thus provides a way to accommodate multiple kinds of cooperation among solvers.

Here's a simple example. Suppose that we have a Wheatstone bridge simulation, with an interactive control for changing the resistance of one of the resistors in the bridge, and a meter showing the output current across the arms of the bridge. There could be one POL (perhaps using constraints, perhaps something else) for handling input events such as moving a control. There is a collection of constraints that specify the physical laws for the simulation (Ohm's Law, Kirchoff's current laws). Finally, there is a graphics part that shows the circuit, meter, and so forth, perhaps with some nice visual effects. We can divide this into three regions: one for handling input, one for simulating the circuit and determining the various voltages and currents, and one for handling the graphic output. These are joined by variables. One variable connects the input POL with the simulation – here the position of the controller on the resistor is read-only as far as the circuit simulation. Another connects the output current to a graphical display of the meter; again, this is read-only for the graphical display. The circuit simulation itself involves some simultaneous linear equations, with an edit variable on one of the resistance values. It can be easily solved by Cassowary. In operation, we repeatedly find a new controller position, then fire up the constraint solver for the bridge, then redraw the graphics.

I had a harder time coming up with a realistic example that uses multiple layers of cooperating solvers.[2] Here is a someone contrived one. Suppose we have something like the above circuit simulation, in terms of an input POL, a simulation, and a graphical display, but instead of a Wheatstone bridge we are animating Quicksort in CLP($\mathcal{R}$), and showing how backtracking produces multiple solutions, which may include variables in the solution with constraints. As noted above, CLP($\mathcal{R}$) uses two cooperating solvers: one for tree constraints and one for constraints over the reals. Solving the Quicksort constraints does require using both solvers in cooperation.

[1] Check this. In particular, are there any problems raised by having read-only variables?

[2] This may mean that it wouldn't arise that often in practice; or perhaps I'm not imagining enough possibilities. In any case, I do like this approach since it dissolves the tension between an ultra-simple way of linking cooperating solvers and selecting among different and more complex schemes.

5 Some Open Questions

The original goal of this note was to suggest an architecture for cooperating constraint solvers; but it seems like we may be able to accommodate other kinds of Problem Oriented Languages, not just constraint languages, at the top level.

- What are the requirements on these POLs?
- What language should solvers be written in?
- How do we handle multiple solutions and cycling among them or otherwise presenting them?
- How do temporal constraints fit into this?
- How often would the multi-layer approach be needed in practice?
- Should we require that the solvers all be sound, that is, that they never give incorrect answers? Or is it OK if they are sometimes heuristic in nature? (I'm reluctant to do that, but perhaps I am being too conservative.)

There are likely numerous other questions that will arise if we push on this further!

References

[1] Alan Borning. *ThingLab—A Constraint-Oriented Simulation Laboratory.* PhD thesis, Stanford, March 1979. A revised version is published as Xerox Palo Alto Research Center Report SSL-79-3 (July 1979).

[2] Alan Borning, Richard Anderson, and Bjorn Freeman-Benson. Indigo: A local propagation algorithm for inequality constraints. In *Proceedings of the 1996 ACM Symposium on User Interface Software and Technology*, pages 129–136, New York, November 1996. ACM.

[3] Alan Borning and Bjorn Freeman-Benson. Ultraviolet: A constraint satisfaction algorithm for interactive graphics. *Constraints*, 3(1):9–32, April 1998.

[4] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992.

[5] Michel Gangnet and Burton Rosenberg. Constraint programming and graph algorithms. In *Second International Symposium on Artificial Intelligence and Mathematics*, January 1992.

[6] Hiroshi Hosobe. A modular geometric constraint solver for user interface applications. In *Proceedings of the 1999 ACM Conference on User Interface Software and Technology*, New York, November 2001. ACM.

[7] Hiroshi Hosobe, Satoshi Matsuoka, and Akinori Yonezawa. Generalized local propagation: A framework for solving constraint hierarchies. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, Heidelberg, Germany, August 1996. Springer-Verlag LLNCS 1118.

[8] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the Fourteenth ACM Principles of Programming Languages Conference*, Munich, January 1987.

[9] Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.

[10] John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[11] Greg Nelson and Derek Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1:245–257, 1979.

[12] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus one-way constraints in user interfaces: Experience with the DeltaBlue algorithm. *Software—Practice and Experience*, 23(5):529–566, May 1993.

[13] Ivan Sutherland. *Sketchpad: A Man-Machine Graphical Communication System*. PhD thesis, Department of Electrical Engineering, MIT, January 1963.