Constraintpropagation in Qualitative Modelling: Domain Variables Improve Diagnostic Efficiency

Bernhard Pfahringer
(email: bernhard@ai-vie.uucp)

Department of Medical Cybernetics and AI
University of Vienna, and
Austrian Research Institute for Artificial Intelligence
Freyung 6, A-1010 Vienna, Austria

September, 1990

Abstract

This paper shows how a specific constraint propagation technique - namely *domain variables* - can speed up qualitative diagnosis considerably. We are using the KARDIO system, a qualitative simulation model of the electrical activity of the heart, to exemplify our points. Furthermore we describe how the domain handling mechanism itself can be implemented in PROLOG efficiently. For a class of applications, where the constraint solver only performs a minor part of the computation our approach is comparable to or better than specialised constraint logic programming systems with regard to overall runtime. Additionally we gain the benefit of being able to specify all of the system in a single language.

Keywords: Constraint Propagation, Unification, Qualitative Modelling, Implementation.

1 Introduction

This paper shows how a specific constraint propagation technique - namely *domain variables* [1] - can speed up qualitative diagnosis considerably. We are using the KARDIO system [2], a qualitative simulation model of the electrical activity of the heart, to exemplify our points. Given a state of the heart (some combination of arrhythmias) the KARDIO model can be used to compute possible ECG patterns and vice versa. The design of the model influences efficiency heavily : simulation (going from arrhythmias to ECG patterns) is fast whereas diagnosis (going from ECG patterns to arrhythmias) is slow. By introducing domain variables the latter can be sped up without changing the structure of the model. Furthermore we will show how to implement all constraint handling in PROLOG itself with good results regarding absolute runtimes.

This paper is outlined as follows: Section 2 introduces the KARDIO model. Section 3 defines domain variables. In section 4 we first show a (naive) implementation in standard PROLOG and then introduce some improvements. Section 5 discusses the results and compares our solution to other approaches.

2 The KARDIO Model

The KARDIO expert system models the electrical activity of the heart in a qualitative way. We will just briefly sketch the model, an extensive description of KARDIO can be found in [2]. Overly simplified, the heart works electrically as follows: certain generators supply electrical impulses which are in turn conducted and combined through specific pathways. These resultant impulses allow the model to predict possible ECG patterns.

The current version of KARDIO relates 943 different combinations of basic arrhythmias to 3096 different ECG patterns yielding a total of 5240 arrhythmia-ECG pairs. Simulation is very efficient: running on an Apollo DN 3000 using compiled QUINTUS PROLOG computing all possible ECGs for a given arrhythmia takes 0.024 seconds on the average. Diagnosis is much slower: computing all possible arrhythmias for a given ECG takes 8.6 seconds on the average. In the following we show how to improve diagnos-

tic efficiency by introducing and combining several constraint propagation techniques.

3 Domain Variables

In this section we briefly introduce *Domain Variables*, a more thorough discussion can be found in [1]. The main idea is simple. Sometimes it is known in advance that a certain variable in a clause should only be bound to one of several possible values. Using standard PROLOG there are just two ways of specifying such knowledge. Either correctness can be tested after the variable has received a value or the set of legal values can be enumerated via backtracking. Both solutions can lead to combinatorial explosion in involved computations. Domain variables allow the variable to be explicitly augmented with the set of legal values, its *domain*. Of course, unification has to be extended to handle such augmented variables. In the following we restrict ourselves to domains consisting solely of atomic values. Furthermore assume that domain variables are represented via a term:

```
domain( FutureValue, LegalValues)
```

Unification has to properly handle three different cases: (1) Unifying a domain variable with a standard variable simply succeeds in binding the standard variable to the domain variable. (2) Unification of a domain variable with an atomic value succeeds with the variable bound to the value, iff the value is legal. (3) The interesting case is unifying two domain variables: both domains have to be intersected to yield the new restricted domain. If the resultant domain contains just one value, the variable can be bound directly to this value. Lastly, if the intersection is empty, unification will simply fail.

Some (experimental) PROLOG systems already support either domains directly [3], whereas others (like Metaprolog [4]) allow for user-defined extensions of the unification algorithm. The latter was available to us, so some test-runs were performed. As the latest version of the KARDIO model was written with domains in mind [5], porting to Metaprolog turned out to be rather simple. Extension of unification for domains can be defined as follows in Metaprolog:

```
:- metafunctor( domain/2 ).

metatermunify( domain(X,Values), X) :- member( X, Values).

metametaunify( domain(X,V1), domain(X,V2) ) :-
   intersection( V1, V2, V3 ),
   new_domain( V3, X).

new_domain( [SingleValue], SingleValue) :- !.
new_domain( [X|L], domain(_,[X|L]) ).
```

First `domain/2` is declared to be a special functor with regard to unification, next unifying a domain variable with a term and unifying two domain variables is specified by appropriate clauses for `metatermunify/2` and `metametaunify/2`. Unification of standard variables with domain variables is handled by the Metaprolog kernel itself. Note that unification of domain variables can result in chains of domain variables. The KARDIO model required just one modification to incorporate domain variables; the original clauses of `mem/2`:

```
mem( X, [X|_]).
mem( X, [_|L]) :- mem( X,L).
```

were replaced by the following clause:

```
mem( X, L) :- X = domain(_,L).
```

The first result - 61 seconds/ECG - was rather discouraging, but should not come as a surprise. Metaprolog is essentially a modified version of the CPROLOG interpreter, whereas the above mentioned average 8.6 seconds were obtained from compiled code. A closer inspection of the way the model is stated revealed that reversing the order of subgoals in the toplevel predicate `heart4` could possibly yield a speed up. For the original (compiled) KARDIO model this hope fails, 22.5 seconds are needed per ECG pattern (see chapter 3.1.4 of [2] for an explanation), but with the advantage of (interpreted) domains to reduce unnecessary backtracking efforts, efficiency is improved to 12.1 seconds/ECG.

The benefit of using domains can also be seen clearly from the total number of subgoal calls (summed up for all 3096 ECG patterns), which is reduced by a bit more than one order of magnitude, from 69 millions down to 4.7 millions. But still absolute runtime is larger for running diagnosis interpreted with domains (12.1 seconds) than running diagnosis compiled without domains (8.6 seconds).

4 Implementing Domain Variables in vanilla PROLOG

The standard way of adding to PROLOG extensions which are themselves implemented in PROLOG is building a meta-interpreter. This technique could of course be used to extend unification by domain variables. But as unification is such a basic operation for a logic programming language, runtime would increase intolerably. Nonetheless we can partially evaluate the application of meta-interpreter to the KARDIO model. To cope with the overwhelming increase in source code size, we have identified three common call patterns that need explicit handling of unification. Before introducing the appropriate predicates to handle these cases, we first show the result of partially evaluating a simple clause which is part of the KARDIO knowledge base:

```
ret_reg_4( _, reg(Loc, _, Rate), reg(Loc, none, zero)) :-
   mem( Rate, [zero, between_250_350, over_350] ).
```

is transformed to:

```
ret_reg_4( _, reg(L1, _, Rate1), reg(L2, Rhythm, Rate2)) :-
   Rhythm := none,
   Rate1 := zero,
   Rate2 <= [zero, between_250_350, over_350],
   L1 :=: L2.
```

This example clause fortunately covers all three cases. Unification of an arbitrary argument (possibly a domain variable) with an atomic value is

handled by `:=`, unification of an arbitrary argument with a list of possible legal values is handled by `<=`, and lastly, unification of two arbitrary arguments is handled by `:=:`.

The predicate `:=/2` is defined as follows:

```
V := Atom :-
   deref( V, Vd ),
   ( var( Vd ) ->
      Vd = Atom
   ;
      atomic( Vd ) ->
         Vd = Atom
   ;
         Vd = domain( Atom, Values ),
         memberchk( Atom, Values )
   ).
```

First, the arbitrary term V is dereferenced, as it could be the starting point of a chain of domain variables. Next, dispatching takes place according to the type of the derefenced value: an unbound variable will get bound to Atom, an atom will be tested for equality, and a domain variable will get bound in the case when membership-testing succeeds.

Derefencing is done by the predicate `deref/2`, which is defined as follows:

```
deref( X, Value) :-
   nonvar( X ), X = domain( Link, _), nonvar( Link),
   !,
   deref( Link, Value).
deref( Value, Value).
```

The first clause detects bound domain variables and dereferences them recursively. The second clause catches all other cases.

The above mentioned predicates `<=/2` and `:=:/2` are defined in a similar straightforward fashion. With these definitions runtime is 7.2 seconds/ECG, a neglegible improvement in efficiency compared to 8.6 seconds/ECG as mentioned in section two. This result can be explained as

follows: backtracking is considerably reduced, but only at the expense of a lot of runtime spent for explicitly unifying terms. After identifying the culprit, remedies can be taken. One well-known principle in computer science is special-case coding of frequently encountered, but easily handled situations. Some statistics gathering revealed the following: most of the time `:=/2`, `<=/2`, and `:=:/2` are called with either unbound variables or atomic values as arguments and not with domain variables. So these predicates had to be modified accordingly to take these special cases into account early. As an example the new definition of `:=/2` is given:

```
V := Atom :-
   ( var(V) ->
      V = Atom
   ;
     atomic(V) ->
      V = Atom
   ;
      deref( V, Vd),
      ( atomic( Vd) ->
         Vd = Atom
      ;
         Vd = domain(Atom,Values),
         memberchk(Atom,Values)
      )
   ).
```

The argument is first of all checked for both of the special cases and if necessary dealt with appropriately, and only then dereferencing and standard dispatching takes place. The other predicates were modified accordingly. This simple and small modification yielded a speedup of a factor of two: 3.4 seconds/ECG.

The next step we undertook was kind of a flow analysis done by hand revealing the following: certain attributes of impulses are never instantiated to or compared with domain variables, namely the `rhythm` of regular impulses and the `focus` of ectopic impulses. The same turned out to be true for most of the variables depicting states of different parts of the heart. Therefore unifications involving only such attributes or variables could be

handled safely and much faster by the builtin unification mechanisms of the underlying PROLOG system. Thus the model was once more automatically transformed to a form as exemplified by our running example `ret_reg_4`:

```
ret_reg_4( _, reg(Loc1, _, Rate1), reg(Loc2, none, Rate2)) :-
   Rate1 := zero,
   Rate2 <= [zero, between_250_350, over_350],
   Loc1 :=: Loc2.
```

Unification of `Rhythm` to `none` is now implicit in the head of the clause. Roughly one third of the number of calls to `:=/2`, `<=/2`, and `:=:/2` were eliminated by this transformation, and the gained efficiency mirrored this figure nicely: runtime per ECG pattern was now down to 2.3 seconds on the average.

So far all the reported transformations were domain-independent, principled ways of improving efficiency. Yet some analysis of the KARDIO model, especially of the toplevel structure, reveals one more source of unnecessary backtracking. When starting from given ECG attributes to generate impulses non-deterministically, these impulses cause backtracking a considerable number of times when fed into the predicates representing the impulse generators of the heart. Chronological backtracking clearly looses in such situations where choices made several subgoals earlier need to be reconsidered. Simple solutions like static or dynamic reordering of subgoal calls did not perform well, either. The former encountered to much interdependencies whereas the latter encurred to much runtime cost. Still there is a simple domain-specific solution to the basic problem. Backtracking efforts can be reduced by supplying minimum constraints for certain impulses beforehand. To stay on principled grounds, all these four impulses were selected, which are directly produced by the generators. For each impulse all solutions were computed (which were few, usually between 5 and 10). Forming the least general generalization for each solution set yielded constraints on the impulses like the following:

```
ImpSA = form( reg( sa_node, _,
                   domain(_,[zero,under_60,between_60_100,
                             between_100_250] )),
              ect( sa_node,no))
```

When incorporating these constraints on the four basic impulses, the gained speedup is an additional factor of almost two: 1.26 seconds/ECG. The following table shortly summarizes the results. *Runtime* is the time needed to find all arrhythmias measured in seconds/ECG, *Calls* is the total number of subgoal calls measured in millions, and *Speedup* is the ratio of the runtimes:

Approach	Runtime	Calls	Speedup
1. Reverse Order	22.54	69.0	1
2. 1 + Domains	7.20	4.7	3
3. 1 + Improved Domains	3.40	4.7	7
4. 3 + Flow Analysis	2.30	4.7	10
5. 4 + Constrained Impulses	1.26	2.5	18

5 Discussion

The following is a comparison of our approach to other possible approaches. Most of the literature on constraint logic programming either argues for, or implicitly assumes that special builtin predicates are necessary for yielding appropriate performance. Therefore available CLP systems are usually prototype PROLOG implementations equipped with extended builtin unification mechanisms like [6] or with additional builtin predicates like [3]. The provided functionality is *opaque* to the user, meaning that the supplied constraint solving method(s) can only be used as is, there is no chance of inspecting, modifying or extending them. And there is a price to be paid: usually execution of the standard PROLOG part of such specialized systems is inferior to commercially available PROLOG systems.

Our example clearly shows that at least one constraint solving technique - *domain variables* - can be implemented in PROLOG itself and still yields results comparable to or better than specialised systems. This should be true not only for the KARDIO model, but also for a larger class of applications exhibiting similar properties. The relatively small domains and their quick reduction to atomic values eliminates the need for special-purpose builtin representation and handling of domains, like what is provided in CHIP [3]. If mechanisms for extending unification as proposed by [4] or [7] find their way into commercial systems, the more elegant approach of

section 3 will possibly be more efficient, too. But till then, our preprocessing way of amalgating PROLOG and domain variables is better off for applications like the KARDIO system.

6 Acknowledgements

I am indebted to Igor Mozetic for providing the KARDIO model, to Christian Holzbaur for providing Metaprolog, to both of them for discussions on the topic, and especially to Robert Trappl for creating a very special working environment. This work was supported by the Austrian Federal Ministry of Science and Research.

References

[1] Hentenryck P.van, Dincbas M.: Domains in Logic Programming, in *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, Morgan Kaufmann, Los Altos, CA, 1986.

[2] Bratko I., Mozetic I., Lavrac N.: *Kardio - A Study in Deep and Qualitative Knowledge for Expert Systems*, MIT Press, Cambridge, MA, 1989.

[3] Hentenryck P.van: *Constraint Satisfaction in Logic Programming*, MIT Press, Cambridge, MA, 1989.

[4] Holzbaur C.: *Realization of Forward Checking in Logic Programming through Extended Unification*, TR-90-11, Oesterreichisches Forschungsinstitut fuer Artificial Intelligence, Wien, 1990.

[5] Mozetic I.: *Diagnostic Efficiency of Deep and Surface Knowledge in KARDIO*, in Artificial Intelligence in Medicine, 2(2), pp.67-83, 1990.

[6] Jaffar J.: *CLP(R) Version 1.0 Reference Manual*, IBM Research Division, T.J.Watson Research Center, Yorktown Heights, N.Y. 10598, 1990.

[7] Neumerkel U.: *Extensible Unification by Metastructures*, Proc. META90, 1990.