A General Relation Constraint: An Implementation

Roman Barták*

Charles University, Faculty of Mathematics and Physics
Department of Theoretical Computer Science and Mathematical Logic
and
Institute for Theoretical Computer Science
Malostranské náměstí 2/25, 118 00 Praha 1, Czech Republic
bartak@kti.mff.cuni.cz
http://kti.mff.cuni.cz/~bartak

Abstract. Constraint Programming is popular for its declarative character that allows users to describe various real-life problems by means of constraints - general relations among several variables. In reality, many constraint solving engines concentrate on tools for working with numerical and logical constraints so general relation constraints should be avoided or they must be expressed in the form of numerical and logical constraints. In the paper we argue for proprietary implementation of the general binary relation constraint. We describe a representation of such constraint and propose several constraint propagation algorithms.

1 Introduction

In theory, the constraint is a general relation among several unknowns but in reality many constraint based systems concentrate on some special forms of constraints, mostly on numerical and logical constraints. This allows the implementers to use dedicated constraint propagation algorithms and thus to increase the efficiency of the constraint satisfaction engine. For example, instead of checking the consistency of numerical constraints pairwise (value by value), the lower and upper bound of the variables' domains are propagated. This dramatically increases the speed of propagation but, as a trade-off, the degree of propagation is slightly lower, i.e., more inconsistent values remain in the domain. Still, dedicated constraint propagation pays off and it is usually recommended to simplify "tabular", i.e. general relation constraints into more mathematical way. Unfortunately, sometimes this conversion cannot be done efficiently and some form of general relation constraint is necessary.

Usually, the easiest way to enter complex real-life constraints is to use a tabular entry form. The user can express arbitrary general relation there that could complicate conversion of such constraint into "mathematical" constraint. The examples of such tabular constraints can be found in planning and scheduling problems where the user expresses transition constraints between activities processed by a resource (see Figure 1) or he/she sets different working times for activities [1,2]. To handle such situations,

* Supported by the Grant Agency of the Czech Republic under the contract no. 201/99/D057.

the constraint programming systems either provide some form of a general binary relation constraint, like `relation` and `element` predicates in SICStus Prolog, or they define an interface for implementation of user-defined constraints, like `dispatch_global` and `fd_global` predicates again in SICStus Prolog [3]. Because the `relation` predicate in SICStus Prolog[1] is not able to handle infinite domains, we decided to implement a more general binary relation constraint using the global constraint programming interface. Surprisingly, though the implementation is pretty straightforward it works well in our problem area - mixed planning and scheduling - and it outperforms even the built-in relation constraint.

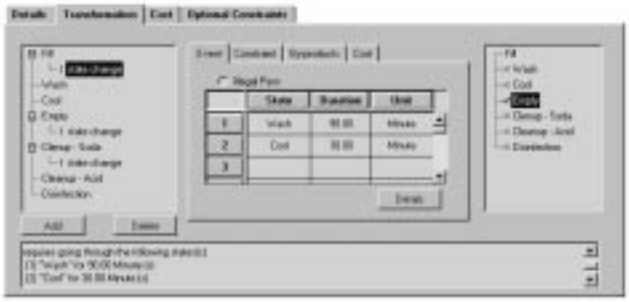

Fig. 1. Tabular input of the transition constraint in the VisOpt System

In the paper we describe the implementation of a general relation constraint in detail. First we define the representation of the general binary relation constraint. Then we describe several constraint propagation algorithms that differ in the ratio between complexity of propagation and memory consumption. The paper is concluded with some final remarks and with the overview of possible ways how to further improve the implementation.

2 Internal Representation

The general relation constraint must be able to capture arbitrary binary relation so the only way, how to represent such constraint is using extensional representation, i.e. an enumeration of compatible or incompatible pairs of values. If some knowledge about the structure of relation is available then it is possible to include some intentional elements into the representation, which decreases the memory consumption and increases the speed of propagation through the constraint.

2.1 Extensional Basis

An extensional representation of a binary relation consists of enumeration of compatible or incompatible pairs of values (in the following text we expect

[1] We are working with SICStus Prolog 3.7.1 that is not the latest release of the system.

enumeration of compatible pairs only)[2]. Naturally, if the domains of related variables are infinite then, in general, such enumeration is infinite as well. In CSP, the infinite domains are used to express unbound variable and usually, after some propagation, the domains become finite so labelling can be performed. Still, it is preferred to allow expressing infinite domains during modelling.

Naturally, we cannot express extensionally the binary constraint where domains of both involved variables are infinite but if the projection of the constraint to one of the variables is finite then we can do it (if one more condition holds - see below). In the following we expect that the projection of the constraint to some variable X is finite, therefore we may expect the domain of X to be finite. Moreover, for each value $a \in dom(X)$ we expect that the projection of the constraint restricted to X=a to the variable Y, the second variable in the constraint, gives a finite number of continuous sets. This needs more explanation. The domain of variable is expected to be a totally ordered set, so it can be represented as a union of disjunctive continuous subsets, where each continuous subset is represented as an interval min..max (min and max are minimum and maximum elements in the subset). For example, the domain of an unbounded variable is simply represented as an interval inf..sup. This mechanism allows us to express many infinite domains extensionally (but not all of them, e.g. even numbers cannot be represented extensionally).

To summarise the above discussion, if the following two conditions hold:

1. the projection of the constraint to one of the constrained variables is finite, lets call this variable X, and
2. for each $a \in dom(X)$ the projection of the constraint restricted to X=a to the other variable (lets call it Y) consists of a finite number of continuous sets

then we can represent the binary constraint extensionally.

We propose to represent a binary constraint (its domain) as a list of pairs a-DY(a), where $a \in dom(X)$ and DY(a) is a nonempty list of disjunctive intervals representing the projection of the constraint restricted to X=a to the domain of Y. The list is sorted using the value *a* in each pair (we expect the domains to be totally ordered); the lists DY(a) are sorted as well in an obvious way. Notice that if above two conditions hold then the list representing the constraint is finite (because of the condition 1) and for every $a \in dom(X)$ DY(a) is finite as well (because of the condition 2).

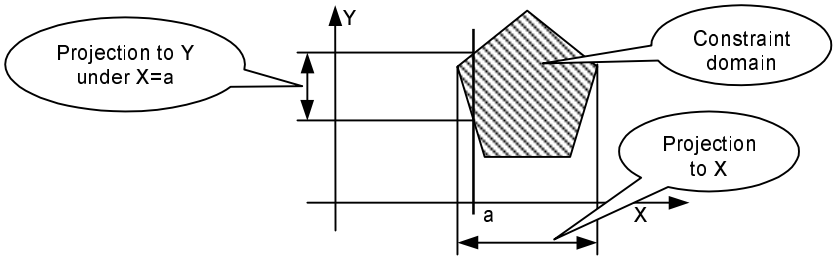

Fig. 2. Projection of the constraint domain

[2] For some constraints, like a disequality constraint, it is more efficient to capture incompatible pairs. Then, the presented propagation algorithms must be adapted to work with the representation using incompatible pairs.

Table 1. Conversion of the tabular form of the relation into the internal representation

Tabular constraint

X	Y	
1	2..20, 30..50	
2	-	*Not compatible*
3	inf..sup	*No restriction on Y*
4	10..50	

Is represented as the list [1-[2..20,30..50], 3-[inf..sup], 4-[10..50]].

Note that the conditions necessary for extensional representation of the constraint are not too restrictive. In practice, the user enters a general relation in the form of a finite table. Consequently, either the domains of both related variables are finite (like in constraints defining transitions between activities during scheduling) or one domain is finite and the other domain (after projection of the constraint) consists of a finite number of intervals.

The proposed representation is almost the same as the representation in SICStus Prolog, but we allow using an infinite domain of the second variable.

2.2 Intentional Elements

If there can be identified some structure of the constraint domain then it is possible to include some intentional information that make the representation more compact and, thus, easier to handle. We have noticed that in many cases the constraints entered by users have the structure that can be collapsed to a rectangle (for example time windows could be the same for all the activities). By an rectangular structure we mean that there exists a subset of the domain of X and a subset of the domain of Y such that all the pairs from the Cartesian product of these subsets are compatible (satisfy the constraint) and all the other pairs are incompatible.

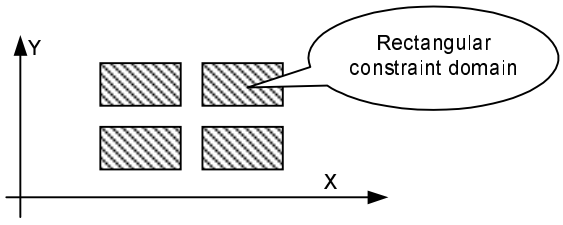

Fig. 3. Rectangular structure of the relation

If we use the above proposed extensional representation then all the DY(x) in the representation are equal. Naturally, we may use a more compact representation that consists of the projections of the constraint to both involved variables. Then the constraint is represented as a single pair DX-DY, where DX and DY are the projection of the constraint to the variable X and Y respectively.

Table 2. Example of tabular constraint with the rectangular structure.

Tabular constraint

X	Y	
1	-	*Not compatible*
2	2..20, 30..50	
3	2..20, 30..50	
4	-	*Not compatible*
5	2..20, 30..50	

is represented as the pair [2..3,5]-[2..20, 30..50] (Note: an interval containing single value is represented as this value).

Though this intentional extension of the extensional representation seems trivial, it can have a large impact on the efficiency. Making this constraint consistent is a one step process as described in the following paragraph.

3 The Propagation Algorithm

Deciding about the representation of the constraint goes hand in hand with the design of a propagation algorithm. The propagation through the constraint ensures that the current domains of X and Y still contain values that are compatible but does not contain any incompatible values. The propagation algorithm can be (locally) *complete* - the domains contain only compatible values after its application, i.e. for every $x \in dom(X)$ there is a compatible value $y \in dom(Y)$ and vice versa (for binary constraints) - or the algorithm can be *incomplete* - it is not guaranteed that all the incompatible values were removed. It may seem curious to use incomplete propagation but in many cases the incomplete propagation is much faster than the complete one while still removing similar quantity of incompatible values. Incomplete propagation is useful if the structure of the constraint is known and, thus the constraint is represented intentionally, e.g. the interval propagation of numerical constraints. However, in case of extensional representation that we use for the general relation constraint, the complexity of complete propagation is almost the same as the complexity of incomplete (e.g., interval) propagation (in both cases the whole constraint domain should be explored) so we decided to implement the complete propagation.

In general, the constraint propagation is called each time a domain of any involved variable is changed. The only exception is so called node consistency, i.e. the propagation through an unary constraint. In such a case, the propagation is done only once because all the values that remain in the domain after the propagation satisfy the constraint. We can apply the same approach to solve "rectangular" relations that are

represented as a pair of compatible domains DX-DY. Then the constraint is disjoined into two unary constraints: (X in DY) and (Y in DY)[3] and it is satisfied "forever".

More complicated situation occurs when the constraint has not a rectangular structure that is a more common case. Then we must go through the constraint domain and remove incompatible values each time the propagation is called[4]. The following algorithm performs the constraint propagation through general binary constraint:

```
1       general_relation_propagation1(Constraint,X,Y)
2          NewDomainOfX <- empty_domain
3          NewDomainOfY <- empty_domain
4          ConstraintDomain <- domain(Constraint)
5          while non_empty(ConstraintDomain) do
6             (x-DY) <- select_and_delete(ConstraintDomain)
7             if x∈domain(X) then
8                CompatibleY <- intersectionOf(domain(Y),DY)
9                if non_empty (CompatibleY) then
10                  NewDomainOfX <- union(NewDomainOfX, {x})
11                  NewDomainOfY <- union(NewDomainOfY, CompatibleY)
12               end if
13            end if
14         end while
15         X in NewDomainOfX
16         Y in NewDomainOfY [see footnote 3]
17      end [if NewDomainOfX or NewDomainOfY is singleton then the
      constraint is entailed and no more propagation is necessary; if
      any domain is empty then the constraint fails]
```

Algorithm 1: Constraint propagation through the general relation constraint

The propagation algorithm exploits the asymmetric representation of the constraint to improve the efficiency - it is faster to test membership of the element in the domain (line 7) ahead of computing the intersection of the domains (line 8) then to do it in the opposite order. Moreover, because the domain of the constraint is ordered we may stop exploring it as soon as we know that the remaining constraint domain consists of the pairs x-DY(x) such that x>max(domain(X)) only. These pairs do not contribute to consistency check. The proposed improvement, the **algorithm 1 enhanced**, differs from the algorithm 1 at line 5 that is replaced by the code:

---

[3] We use the notation of SICStus Prolog where (X in DX) means the variable X has the domain DX. If any domain has already been assigned to X then an intersection with the domain DX is used as a new domain of X.

[4] Usually, propagation through the constraint is called when the domain of any involved variable is changed. Sometimes, the propagation is evoked only when minimum or maximum of the domain is changed (useful when interval propagation is used).

```
18          while min{x|(x-DY)∈ConstraintDomain} ≤ max(domain(X)) do
```

Naturally, to exploit fully this improvement we recommend selecting and deleting the elements from the constraint domain in the order defined for the representation (smaller x first). This is natural, if the domain is represented as an ordered list.

3.1 Memory Consumption vs. Speed of Constraint Propagation

During each call to the propagation procedure, (almost) the whole constraint domain is explored. It seems more efficient that if the domain of variables is changed then we also change the domain of the constraint by removing the elements incompatible with current domains of the variables. Then, we will remember only the part of the constraint domain that is relevant to current domains of the variables and therefore exploring this domain is faster during propagation. The constraint domain is relevant to the domains of variables if for each (x-DY) in the constraint domain the formula ($x \in dom(X)$ & $DY \subseteq dom(Y)$) holds. The following propagation algorithm maintains the constraint domain relevant to the domains of the variables.

```
19      general_relation_propagation2(Constraint,X,Y)
20         NewDomainOfX <- empty_domain
21         NewDomainOfY <- empty_domain
22         NewConstraint <- empty_domain
23         ConstraintDomain <- domain(Constraint)
24         while min{x|(x-DY)∈ConstraintDomain} ≤ max(domain(X)) do
25            (x-DY) <- select_and_delete(ConstraintDomain)
26            if x∈domain(X) then
27               CompatibleY <- intersectionOf(domain(Y),DY)
28               if non_empty (CompatibleY) then
29                  NewDomainOfX <- union(NewDomainOfX, {x})
30                  NewDomainOfY <- union(NewDomainOfY, CompatibleY)
31                  NewConstraint <- (x-CompatibleY) : NewConstraint
32               end if
33            end if
34         end while
35         X in NewDomainOfX
36         Y in NewDomainOfY
37         Constraint <- revert(NewConstraint)
38      end
```

Algorithm 2: Constraint propagation that changes the constraint domain

There is one problem with updating the domains of variables and constraints during constraint propagation. All the changes should be backtrackable because the

constraint propagation can be evoked during the labelling stage where alternative values are tried to find the solution and backtracking is used when a failure occurs. If copying of domains is used (and this is the case of implementation of FD solvers in most Prolog systems) then we need a lot of memory to remember changes of the constraint domain. This is necessary because both the list representing the constraint domain and the domains DY are changed (see Figure 4).

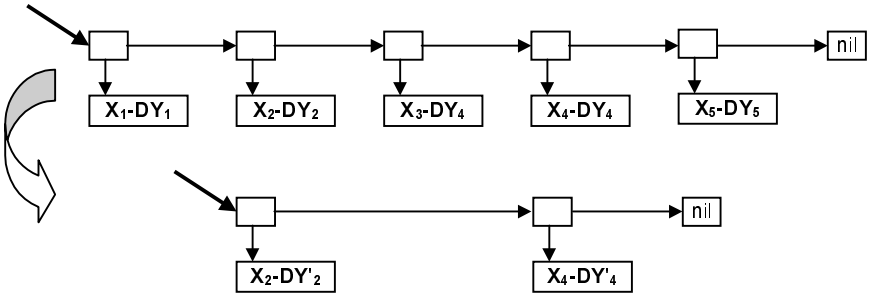

Fig. 4. Constraint domain before (top) and after (down) the constraint propagation using algorithm 2. The list is represented as a one-way pointer structure.

It seems that especially the changes of domains DY are memory consuming so we propose the following compromise: remember only the changes of the domain of the variable X and leave the domain DY unaltered. This strategy can be implemented by substituting the line 31 in the algorithm 2 by the following code (**algorithm 2 enhanced**):

```
39          NewConstraint <- (x-DY) : NewConstraint
```

The resulting cardinality of the constraint domain measured as the length of the representation is equal to the size of the domain X after propagation. Because the elements of the representation do not change during the propagation, we can use a list of pointers to pairs (x-DY) instead of copying the complete representation. This is the way of handling lists in Prolog systems so in our implementation this behaviour is inherited from Prolog.

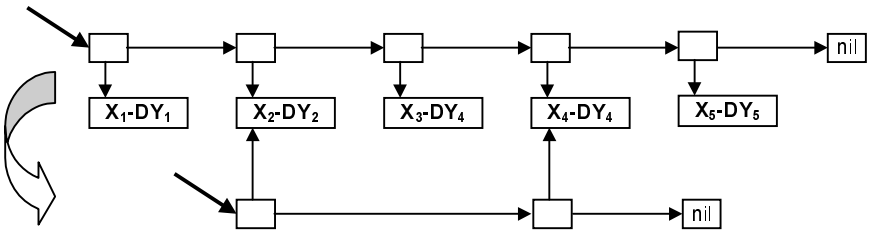

Fig. 5. Constraint domain before (top) and after (down) the constraint propagation using algorithm 2 enhanced. Both representations share the same elements.

Unfortunately, when this implementation of the general relation constraint was used in a real-life scheduling problem [1,2] we noticed that the memory consumption is

still too big so large-scale problems cannot be solved completely. Therefore we decided to minimise the changes to the constraint domain during each propagation step. In fact, we propose not to change the constraint domain at all but to move only the pointer indicating the first element of the constraint domain. Thus if the minimum value of the domain of the value X is changed then the pointer indicating the beginning of the respective constraint domain is changed accordingly.

```
40      general_relation_propagation3(Constraint,X,Y)
41         NewDomainOfX <- empty_domain
42         NewDomainOfY <- empty_domain
43         NewConstraint <- nil
44         ConstraintDomain <- domain(Constraint)
45         while min{x|(x-DY)∈ConstraintDomain} ≤ max(domain(X)) do
46            (x-DY) <- select_and_delete(ConstraintDomain)
47            if x∈domain(X) then
48               CompatibleY <- intersectionOf(domain(Y),DY)
49               if non_empty (CompatibleY) then
50                  NewDomainOfX <- union(NewDomainOfX, {x})
51                  NewDomainOfY <- union(NewDomainOfY, CompatibleY)
52                  if NewConstraint = nil then
53                     NewConstraint <- ConstraintDomain
54                  end if
55               end if
56            end if
57         end while
58         X in NewDomainOfX
59         Y in NewDomainOfY
60         Constraint <- NewConstraint
61      end
```

Algorithm 3: Constraint propagation with shallow memory

The algorithm 3 does not change the domain of the original constraint it just moves the pointer indicating the beginning of the current constraint domain as Figure 6 shows. This is useful if there are a lot of constraints sharing the same domain; e.g., the constraint describing permissible transitions is applied to each pair of consecutive activities (then each constraint has its own pointer to common data structure). Naturally, this is much more memory efficient than using algorithm 2 that generates a new constraint domain when there is any change during propagation[5].

[5] The behaviour of the algorithm 2 corresponds to the `element` constraint in SICStus Prolog that also saves all the changes of DY domains. Moreover the `element` constraint requires the projection of the constraint domain to variable X to be an interval starting from 1.

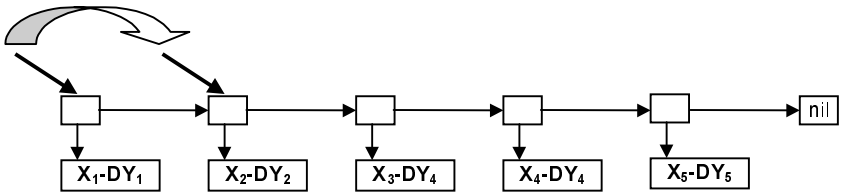

Fig. 6. Constraint domain before (left pointer) and after (right pointer) the constraint propagation using algorithm 3. Only the pointer is changed as the arrow indicates.

The trade-off of the algorithm 3 is checking some pairs of values repeatedly even if it is known that they are incompatible. Table 3 compares memory consumption and complexity of all proposed algorithms (upper estimates). By memory consumption we mean memory requirements of single propagation step additional to the representation of the original constraint domain (thus, the algorithm 1 consumes no additional memory while the algorithm 2 duplicates the constraint domain). We measure the algorithm complexity using the number of value pairs explored in single propagation step.

Table 3. Comparison of memory consumption and complexity of proposed algorithms.

Algorithm	Memory consumption Complexity of single propagation step
1	0 O(|original_domain(X)|*|original_domain(Y)|)
1 enhanced	0 O(|(original_min(X))..current_max(X)) ∩(original_domain(X))|*|original_domain(Y)|)
2	O(|current_domain(X)|*|current_domain(Y)|) O(|current_domain(X)|*|current_domain(Y)|)
2 enhanced	O(|current_domain(X)|) O(|current_domain(X)|*|original_domain(Y)|)
3	O(1) O(|(current_min(X)..current_max(X))∩(original_domain(X))|*|original_domain(Y)|)

4 Conclusions

In the paper we describe an implementation of general binary relation constraint and we discuss the trade-off between memory consumption and speed of constraint propagation steps. The proposed algorithms were implemented in SICStus Prolog 3.7.1 and used in the scheduling engine for complex process environments. We did not performed extensive comparison with build-in `relation` predicate yet partly because in the project we use relation constraints with infinite domains that are not supported by the build-in `relation` predicate in SICStus Prolog. However, the first experiments show significant increase of speed caused by presence of "rectangular" constraints that are handled intentionally.

Our experiments with the relation constraint and the known wisdom of using mathematical constraints instead of tabular ones confirm that the intentional

representation of the constraint domain increases dramatically the speed of constraint propagation because we do not need to check all the value pairs (in the binary constraints). Therefore our next research is directed to identifying other special structures in the constraint domain that could be encoded intentionally to increase the speed of constraint propagation and decrease memory consumption.

In the paper we concentrate on the propagation algorithms for binary constraints so the open question is whether these algorithms can be modified to work with N-ary constraints for N>2. In real-life applications we can identify ternary constraints, e.g., to capture transition time between two activities, that can still be inputted in the tabular form (see Figure 1). We model this constraint using binary constraint but we believe that proprietary implementation of ternary relation constraint could be more efficient.

Acknowledgements

Author's work is supported by the Grant Agency of the Czech Republic under the contract number 201/99/D057 and by InSol Ltd. I would like to thank François Laburthe and referees of the paper for useful remarks.

References

[1] Barták, R. *Dynamic Constraint Models for Planning and Scheduling Problems*. In New Trends in Constraints (Papers from the Joint ERCIM/CompulogNet Workshop, Cyprus, October 25-27, 1999), LNAI 1865, Springer Verlag, 2000.

[2] Barták, R. *VisOpt – the Solver behind the User Interaction*. White Paper, InSol Ltd., Israel, 1999.

[3] Carlsson M., Ottosson G., Carlson B. *An Open-Ended Finite Domain Constraint Solver*. In Proc. Programming Languages: Implementations, Logics, and Programs, 1997.

[4] Sterling L., Shapiro E. *The Art of Prolog*. The MIT Press, Cambridge, Massachusetts, 1986.

[5] Tsang E. *Foundations of Constraint Satisfaction*. Academic Press, London, 1995.