INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
Oettingenstraße 67, D–80538 München

Ludwig
Maximilians–
Universität
München
LMU

Constraint Handling Rules: Applications and Extensions

Slim Abdennadher

The 12th International Conference on Applications of Prolog, INAP'99
http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-1999-8, August 1999

Constraint Handling Rules: Applications and Extensions

Slim Abdennadher
Computer Science Department, University of Munich
Oettingenstr. 67, 80538 München, Germany
Slim.Abdennadher@informatik.uni-muenchen.de

1 Introduction

The success of constraint programming in expressing and solving hard real-world problems has created an increasing demand for more flexible and application-oriented customization of constraint solvers. The declarative language Constraint Handling Rules (CHR) [Frü98] is a successful proposal especially designed for the executable specification of user-defined constraints.

In this paper we present a finite domain solver written in CHR, which performs hard and soft constraint propagation. Hard constraints are conditions that must be satisfied, soft constraints, however, may be violated, but should be satisfied as much as possible. The solver is powerful enough to serve as the core of a university timetabling system.

Then we show that a small and simple extension to CHR makes it a general-purpose constraint logic programming language. The extended language, called "CHR$^\vee$", turns out to be a very flexible query language since it supports several (constraint) logic programming paradigms and allows to mix them in a single program. In particular, it supports top-down query evaluation and also bottom-up evaluation as it is frequently used in (disjunctive) deductive databases.

2 Constraint Handling Rules

Constraint Handling Rules (CHR) [Frü98] is a declarative high-level language extension especially designed for writing constraint solvers. With CHR, one can introduce *user-defined* constraints into a given host language, be it Prolog, Lisp or any other language.

2.1 Syntax and Semantics

A *constraint* is a first order atomic formula. We use two disjoint sorts of predicate symbols for two different classes of constraints: One sort for *built-in* constraints and one sort for *user-defined* constraints. Built-in constraints are those handled by a predefined constraint solver that already exists. User-defined constraints are those defined by a CHR program.

A *CHR program* is a finite set of rules. There are two basic kinds of rules.

A *simplification rule* is of the form $H \Leftrightarrow C \mid B$ and a *propagation rule* is of the form $H \Rightarrow C \mid B$, where the *head* H is a non-empty conjunction of user-defined constraints, the *guard* C is a built-in constraint and the *body* B is a conjunction of built-in and user-defined constraints. A guard "*true*" is usually omitted together with the vertical bar.

The operational semantics of CHR can be described as a state transition system for *states* of the form G, where G (the *goal*) is a conjunction of user-defined and built-in constraints.

Given a CHR program P we define the transition relation $\mapsto_P$ by introducing three kinds of computation steps (Figure 1).

In Figure 1, the notation G_{built} denotes the built-in constraints in a goal G. An equation $c(t_1,\ldots,t_n){=}d(s_1,\ldots,s_n)$ of two constraints stands for $t_1{=}s_1 \wedge \ldots \wedge t_n{=}s_n$ if c and d are the same predicate symbols and for *false* otherwise. An equation $(p_1 \wedge \ldots \wedge p_n){=}(q_1 \wedge \ldots \wedge q_m)$ stands for $p_1{=}q_1 \wedge \ldots \wedge p_n{=}q_n$ if $n = m$ and for *false* otherwise. Conjuncts can be permuted since conjunction is associative and commutative.

Solve

If	$CT \models \forall\, (G_{built} \leftrightarrow G'_{built})$
and	G' is the "normal form" of G
then	$G \mapsto_P G'$

Simplify

If	$(H \Leftrightarrow C \mid B)$ is a rule with variables $\bar{x}$
and	$CT \models \forall\, (G_{built} \rightarrow \exists\bar{x}(H{=}H' \wedge C))$
then	$(H' \wedge G) \mapsto_P (H{=}H' \wedge B \wedge C \wedge G)$

Propagate

If	$(H \Rightarrow C \mid B)$ is a rule with variables $\bar{x}$
and	$CT \models \forall\, (G_{built} \rightarrow \exists\bar{x}(H{=}H' \wedge C))$
then	$(H' \wedge G) \mapsto_P (H{=}H' \wedge B \wedge C \wedge H' \wedge G)$

Figure 1: Computation Steps of CHR

In the **Solve** computation step, the built-in solver normalizes the built-in constraints appearing in the state G. To normalize the built-in constraints means to produce a new state G' that is (according to the constraint theory CT) logically equivalent to G.

To **Simplify** user-defined constraints H' means to remove them from the state $H' \wedge G$ and to add the body B of a fresh variant of a simplification rule ($H \Leftrightarrow C \mid B$) and the equation $H{=}H'$ and the guard C to the resulting state G, provided H' matches the head H and the guard C is implied by the built-in constraints appearing in G. In this case we say that the rule R is *applicable to* H'. A "variant" of a formula is obtained by renaming its variables. "Matching" means that H' is an instance of H, i.e. it is only allowed to instantiate (bind) variables of H but not variables of H'. In the logical notation this is achieved by existentially quantifying only over the fresh variables $\bar{x}$ of the rule to be applied in the condition.

The **Propagate** transition is like the **Simplify** transition, except that it keeps the constraints H' in the state. Trivial nontermination caused by applying the same propagation rule again and again is avoided by applying a propagation rule at most once to the same constraints. A more complex operational semantics that addresses this issue can be found in [Abd97].

2.2 CHR by Example

Finite domains appeared first in CHIP [vH89] by incorporating consistency algorithms [Mac77, Mac92] into constraint logic programming. Implementing these techniques with CHR is straightforward. The constraint `X::Dom` means that the value for the variable `X` must be in the given finite domain `Dom`.

The finite domain solver contains rules like:

```
X::[] ⇔ false.

X::L1 ∧ X::L2 ⇔
    intersection(L1,L2,L3) ∧ X::L3.
```

The first rule reads: Replace the constraint `X::[]` by the constraint `false` exhibiting its inconsistency. The second rule intersects two domains for the same variable, thus tightening the domain.

The constraints `X::[2,3,4]` and `X::[5,6]` can be simplified to `X::[]` by the second rule. This constraint in turn simplifies to `false` with the first rule, so that the inconsistency of the initial constraints is exhibited.

3 Applications

Constraint Handling Rules have been used successfully in challenging applications, where other existing constraint logic programming systems could not be applied with the same results in terms of simplicity, flexibility and efficiency. In the following, we present a constraint solver written in CHR to solve the university course timetabling problem.

University course timetabling problems are combinatorial problems which consist in scheduling a set of courses within a given number of rooms and time periods. Solving a real world timetabling problem manually often requires a significant amount of time, sometimes several days or even weeks since several constraints must be taken into account.

Timetabling the courses offered at the Computer Science Department of the University of Munich requires the processing of hard and soft constraints. Hard constraints are conditions that must be satisfied, soft constraints, however, may be violated, but should be satisfied as much as possible. In practice, most constraint-based timetabling systems either do not support soft constraints [AB94] or use a branch & bound search instead of chronological backtracking [HW95, FHS95]. Another approach is to adopt techniques developed to propagate hard constraints; soft constraint propagation is intended to associate values with an estimate of how selecting a value will influence solution quality, i.e. which value is known (or expected) to violate soft constraints, or the other way round, which value is known (or expected to) satisfy soft constraints. By considering estimates in value selection, one hopes that the first solution will satisfy a lot of soft constraints.

Our aim is to implement a similar approach for our timetabling problem using CHR [AM98, AM99]. The classical finite domain solver presented above is not sufficient for our needs: Since soft constraints may be violated, the values to be constrained must not be removed from the variable's domain. Moreover, when we have to choose a value for the variable during search, we must be able to decide whether a certain value is a good choice or not. Therefore, each value must be associated with an assessment. We chose to represent a domain as a list of value-assessment pairs. For example, assume the domain of `X` is `[(3, 0), (4, 1), (5, -1)]`. Then `X` may take one of values `3`, `4` and `5`, whereas `4` is encouraged with assessment `1` and `5` is discouraged with assessment `-1`.

The solver is based on three types of constraints.

- `domain(X,D)` means that `X` must get assigned a value occuring in the list of value-assessment pairs `D`.
- `in(X,L,W)`: Its meaning depends on the weight `W`. If `W = infinite`, i.e. if the constraint is hard, it means that `X` must not get assigned a value not occuring in the list `L`. If `W` is a number, i.e. if the constraint is soft, it means that the assessment for the values occuring in `L` and not yet removed from the domain of `X` should be increased by `W`.
- `notin(X,L,W)`, if hard, means that `X` must not get assigned any of the values occuring in the list `L`. If it is soft, it means that the assessment for the values occuring in `L` and not yet removed from the domain of `X` should be decreased by `W`.

The Solver's Core

Propagating a soft constraint is intended to modify the assessment of the values to be constrained. For example, assume the existence of the constraint `in(X, [3], 2)` stating that `3` should be assigned to `X`. Then we have to increase the assessment for value `3` in the domain of `X` by adding `2` to the current assessment of `3` obtaining the new domain `[(3, 2), (4, 1), (5, -1)]` for `X`. However, applying a hard constraint will still mean to remove values from the variable's domain. Consequently, an `in` constraint is processed by either pruning the domain or increasing the assessment for the given values.

```
domain(X,D) ∧ in(X,L,W) ⇔ W = infinite |
    domain_intersection(D,L,D1) ∧
    domain(X,D1).

domain(X, D), in(X,L,W) ⇔ W ≠ infinite |
    increase_assessment(W,L,D,D1) ∧
    domain(X,D1).
```

Whenever an `in` constraint arrives, the rules look for the corresponding `domain` constraint and replace both by a new `domain` constraint containing a modified domain. For a hard `in` constraint, i.e. in case `W = infinite`, this is the intersection of the domain `D` with the list of values `L`. For a soft `in` constraint this is `D` with the assessment for the values occuring in `L` increased by `W` resulting in `D1`. The rules for `notin` are similar.

```
domain(X,D) ∧ notin(X,L,W) ⇔ W = infinite |
    domain_subtraction(D,L,D1) ∧
    domain(X,D1).

domain(X,D) ∧ notin(X,L,W) ⇔ W ≠ infinite |
    decrease_assessment(W,L,D,D1) ∧
    domain(X,D1).
```

Subtracting weights, which are always positive, may result in negative assessments.

Whenever a variable's domain has been reduced to the empty list, the variable cannot get assigned a value without violating hard constraints. This case is dealt with by the following simplification rule.

```
domain(_,[]) ⇔ false.
```

With only one value left in a variable's domain we can assign the remaining value to the variable immediately.

```
domain(X,[(A,_)]) ⇒ X = A.
```

We use a propagation rule instead of a simplification rule because the `domain` constraint must not be removed. Without it the processing of `in` and `notin` constraints imposed on the variable's domain would not be guaranteed and thus an inconsistency might be overlooked.

4 Extensions

In contrast to logic programming languages, CHR is a committed-choice language. That is, whenever more than one transition is possible, one transition is chosen nondeterministically (in the sense of don't-care nondeterminism, i.e., without backtracking). Furthermore, CHR performs a one-sided unification ("matching") between goals and rule heads rather than unification. These incompatibilities make CHR difficult to use as a general-purpose logic programming language, especially for search-oriented problems. In the following, we show that a small and simple extension to CHR makes it a general-purpose constraint logic programming language: We allow disjunctions on the right hand sides of CHR rules.

So we extend the syntax of CHR: In a simplification rule $H \Leftrightarrow C \mid B$ the body B is now a formula that is constructed from atoms by conjunctions and disjunctions in an arbitrary way. For the sake of uniformity the syntax of propagation rules is extended in the same way and also goals may be of the same form as a rule body. We call the extended language "CHR$^{\vee}$" [AS98].

The computation steps from Figure 1 will be used for CHR$^{\vee}$ programs in the same way as they have been used for CHR programs. However, with the extended syntax disjunctions make their way into the state. In order to handle these, we introduce the **Split** computation step (Figure 2).

Split
$$(G_1 \vee G_2) \wedge G \mapsto G_1 \wedge G \mid G_2 \wedge G$$

Figure 2: Computation step for disjunctions

The **Split** step can always be applied if the state contains a disjunction. No other condition needs to be satisfied. The step leads to branching in the computation. So we get a tree rather than a chain of states. In the two children of a state containing a disjunction this disjunction will be replaced by one or the other alternative of the disjunction, respectively.

Any Horn clause program can be converted into an equivalent CHR$^{\vee}$ program. The central idea is to move non-committed choices and unification to the right hand sides of rules. The required transformation turns out to be Clark's completion of logic programs [Cla78].

Consider for example the well-known ternary `append` predicate for lists, which holds if its third argument is a concatenation of the first and the second argument. It is usually implemented by these two Horn clauses:

```
append([],L,L) ← true.
append([H|L1],L2,[H|L3]) ← append(L1,L2,L3).
```

The corresponding CHR$^{\vee}$ program consists of the single simplification rule

```
append(X,Y,Z) ⇔
 (      X=[] ∧ Y=L ∧ Z=L
 ∨      X=[H|L1] ∧ Y=L2 ∧ Z=[H|L3] ∧
         append(L1,L2,L3) ).
```

$\mathrm{CHR}^{\vee}$ supports a new programming style where top-down evaluation, bottom-up evaluation, and constraint solving can be intermixed in a single language. Examples for mixing logic programming paradigms using $\mathrm{CHR}^{\vee}$ can be found in [AS98].

5 Conclusion

In this paper, we have argued that CHR is a good vehicle for implementing application-oriented constraint solvers to solve hard real-world problems. Simplicity, flexibility, efficiency, and rapid prototyping are the advantages of using CHR.

Furthermore, we have presented the language $\mathrm{CHR}^{\vee}$, a simple extension of Constraint Handling Rules. $\mathrm{CHR}^{\vee}$ introduces disjunctions into the bodies of rules and into the goals. We have seen that $\mathrm{CHR}^{\vee}$ retains the extra features of CHR, e.g., committed choice and matching, by performing the unification on the right hand side of a rule and by doing backtracking within a single rule rather than between rules. With this programming style, several paradigms, e.g. top-down evaluation, bottom-up evaluation and constraint solving can be intermixed in a single language.

References

[AB94] F. Azevedo and P. Barahona. Timetabling in constraint logic programming. In *Proceedings of 2nd World Congress on Expert Systems*, Estoril, Portugal, January 1994.

[Abd97] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Third International Conference on Principles and Practice of Constraint Programming*, LNCS 1330. Springer, 1997.

[AM98] S. Abdennadher and M. Marte. University timetabling using constraint handling rules. In *Actes des Journées Francophones de Programmation en Logique et Programmation par Contraintes*, 1998.

[AM99] S. Abdennadher and M. Marte. University timetabling using constraint handling rules. *Journal of Applied Artificial Intelligence, Special Issue on Constraint Handling Rules*, 1999. To appear.

[AS98] S. Abdennadher and H. Schütz. $\mathrm{CHR}^{\vee}$: A flexible query language. *Flexible Query Answering Systems*, LNAI 1495, 1998.

[Cla78] K. Clark. *Logic and Databases*, chapter Negation as Failure, pages 293–322. Plenum Press, 1978.

[FHS95] H. Frangouli, V. Harmandas, and P. Stamatopoulos. UTSE: Construction of optimum timetables for university courses — A CLP based approach. In *Proceedings of the Third International Conference on the Practical Applications of Prolog (PAP'95)*, pages 225–243, Paris, France, April 1995.

[Frü98] T. Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, 1998.

[HW95] M. Henz and J. Wurtz. Using Oz for college time tabling. In *Proceedings of the First International Conference on the Practice and Theory of Automated Timetabling (ICPTAT '95)*, pages 283–296, 1995.

[Mac77] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Mac92] A. K. Mackworth. Constraint satisfaction. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 285–293. Wiley, 1992. Volume 1, second edition.

[vH89] P. van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, Massachusetts, 1989.