Rule-based constraint propagation : theory and applications
Brand, S.

Citation for published version (APA):
Chapter 2
Rule-Based Constraint Programming

Rule-based constraint programming means the adoption of a rule-based approach to solving constraint satisfaction problems. In this chapter, we first introduce constraint programming and then discuss rule-based programming. We proceed by discussing the application of this paradigm to constraint programming, before coming to what we regard as its most relevant aspect, namely rule-based constraint propagation.

2.1 Constraint Programming

2.1.1 Overview

Constraint programming is an alternative approach to programming in which a problem is first modelled declaratively and then solved by general or domain-specific methods. See [Tsang, 1993, Marriott and Stuckey, 1998, Apt, 2003, Dechter, 2003, Frühwirth and Abdennadher, 2003], for instance.

We begin with an overview; the formal framework is introduced in the following sections.

Modelling. A problem model in constraint programming consists of requirements — constraints — on variables so that acceptable variable assignments correspond to solutions to the problem. The variables have domains, i.e. sets of possible values. A constraint is a relation that specifies which combination of domain values is acceptable; it can be defined extensionally or intensionally. A problem formulated in this way is called a constraint satisfaction problem (CSP).

The CSP framework is expressive; many computationally intractable problems can immediately be formulated as CSPs. In a 3-SAT problem, each clause constrains three propositional variables. In graph colouring, where different colours must be assigned to connected vertices, the constraints are simple disequalities. Combinatorial problems often have simple formulations as CSP. A strength of
modelling with constraints is its flexibility due to the compositional nature of CSPs: individual constraints can be changed, added or removed disregarding the rest of the problem.

Furthermore, problems exist that do not directly correspond to CSPs but contain CSPs as subproblems. For instance, optimisation problems can often be naturally decomposed into constraints describing what a solution is, and an objective function rating the quality of a solution. This leads to the concept of a *constraint optimisation problem*. In other situations, a problem gives rise to a *sequence* of CSPs. An example of this is planning. The problem whether a plan of a given length exists transforming one state into another can typically be represented as a CSP. Shortest plans can be found by searching for fixed-length plans and iteratively increasing the plan length, which means repeatedly trying to solve a CSP.

**Solving.** For some specific CSP classes, specialised solution methods exist. A system of linear equations induces a CSP in which every equation is a constraint, and which can be solved well by Gaussian Elimination, for example. Term unification can be viewed as a CSP in which the variables range over some term universe; a unification algorithm solves such a problem.

A general method to find solutions of CSPs is systematic *search*: splitting the problem into more specific subproblems that are examined separately. Backtracking is a commonly used algorithm for this purpose.

The search space can be reduced by *constraint propagation*. The principle of constraint propagation is the *controlled deduction* of new constraints. The derived constraints are added to the problem so as to make its solutions *more explicit* without changing them. The options for constraint propagation include, on the one hand, which or how many currently explicit constraints to consider at a time, and on the other hand, what constraints to deduce. The identification of the types of constraint propagation that are both useful and have acceptable computational cost and the development of corresponding efficient *constraint propagation algorithms* are important issues in constraint programming research.

The result of complete constraint propagation is typically characterised by a *local consistency* notion. Generally, a local consistency only approximates global consistency (concerning the entire CSP), and usually the two are incomparable in the sense that none entails the other.

Since a CSP model of a problem does not prescribe the solution method, alternatives to systematic search are possible. Notably, *local search* turns out to perform very well on some types of CSPs. In local search, a total variable assignment — a solution *candidate* — is considered. If not all constraints are satisfied, the assignment is modified, and the process is repeated. The changes to the assignment are mostly local and controlled by heuristics. For instance, a variable is selected whose assigned value violates a constraint, and the assigned
value is replaced.
In contrast to systematic search, local search is incomplete: it is not ensured that all possible assignments (i.e. the complete search space) are visited. Hence, local search is unusable for proving unsatisfiability, nor is it guaranteed to find a solution of a satisfiable problem.

A brief history. Constraint programming has its roots in the field of constraint satisfaction in Artificial Intelligence in the 1970s; see for example [Montanari, 1974, Kumar, 1992]. In the 1980s, these techniques were connected with the declarative problem solving approach of logic programming, which resulted in constraint logic programming [Jaffar and Maher, 1994]. More recently and on-going, a fruitful import of techniques notably from Operations Research has broadened the scope of the field. Applications of constraint programming include various problems in Artificial Intelligence (temporal reasoning, various forms of spatial reasoning) and Combinatorial Optimisation (scheduling, resource allocation, configuration, planning). Industrial interest in constraint programming techniques continues to provide a significant impetus.

2.1.2 Constraint Satisfaction Problems
We now introduce constraint programming formally.

Constraints
Consider a finite sequence of different variables

\[ X = x_1, \ldots, x_m \]

with respective domains

\[ D_1, \ldots, D_m, \]

so each \( x_i \) takes its value from the set \( D_i \). A constraint \( C \) on \( X \) is a pair

\[ (C_R, X). \]

\( C_R \) is an \( m \)-ary relation and a subset of the Cartesian product of the domains,

\[ C_R \subseteq D_1 \times \cdots \times D_m. \]

The elements of \( C_R \) are the solutions of the constraint, and \( m \) is its arity. Nothing more is stipulated about \( C_R \); in particular, it can be defined intensionally and it can be infinite.

It is useful to mention two special cases of constraints. In the true constraint, we find \( C_R = D_1 \times \cdots \times D_m \), while \( C_R = \emptyset \) in the false constraint. These are the only two cases in which we admit \( m = 0 \), whereas we generally require \( m \geq 1 \).

We sometimes write \( C(X) \) for the constraint and often identify \( C \) with \( C_R \).
2.1.1. Example. Here are some constraints and their variables.
\[
\begin{align*}
\langle \{(A, B), (C, A), (B, C)\}, \{x, y\}\rangle, & \quad x, y \in \{A, B, C\}; \\
\langle \{(a, b, c, n) \in \mathbb{N}^4 \mid a^n + b^n = c^n\}, \{x, y, z, u\}\rangle, & \quad x, y, z, u \in \mathbb{N}; \\
\langle \{(p_1, \ldots, p_m) \in \{0, 1\}^m \mid \exists k. p_k = 1\}, \{x_1, \ldots, x_m\}\rangle, & \quad x_i \in \{0, 1\} \text{ for all } i.
\end{align*}
\]
\(\mathbb{N}\) is the set of natural numbers.

Constraint Satisfaction Problems

A constraint satisfaction problem, in short CSP, consists of a finite sequence of variables \(X = x_1, \ldots, x_n\) with respective domains \(D = D_1, \ldots, D_n\), and a finite set \(C\) of constraints, each on a subsequence of \(X\). A CSP can thus be viewed as a triple
\[
(C, X, D).
\]
We use also the notational variant \(\langle C; x_1 \in D_1, \ldots, x_n \in D_n \rangle\).

Solutions

Consider some variable sequence \(X = x_1, \ldots, x_n\) and an element \(d = d_1, \ldots, d_n\) of the product of variable domains \(D_1 \times \cdots \times D_n\). By the projection of \(d\) on a subsequence \(Y = x_{i(1)}, \ldots, x_{i(\ell)}\) of \(X\), we mean the sequence \(d_{i(1)}, \ldots, d_{i(\ell)}\) which we denote by \(d[Y]\). In particular, we have \(d[x_k] = d_k\). Lifting this notion to constraints, we write \(C[Y]\) for the set \(\{d[Y] \mid d \in C_R\}\) where \(C = (C_R, X)\).

By a solution to the CSP \(\langle C, X, D \rangle\) we mean an element \(d \in D_1 \times \cdots \times D_n\) such that for each constraint \(C \in C\) on a sequence of variables \(Y\) we have \(d[Y] \in C\). We call a CSP consistent if it has a solution. The set of all solutions of a CSP \(\mathcal{P}\) is denoted by \(\text{Sol}(\mathcal{P})\).

2.1.2. Example. Consider the CSP
\[
\mathcal{P} = \langle \{\text{fermat}(x, y, z, n), \text{even}(y)\}; \ x, y, z, n \in \mathbb{N}\rangle.
\]
The constraint \(\text{fermat}(x, y, z, n)\) is \(x^n + y^n = z^n\), and even is the set of even integers. \(\mathcal{P}\) is consistent: the tuple \(d = (3, 4, 5, 2)\) is a solution. Indeed, we have
\[
\begin{align*}
d[x, y, z, n] &= (3, 4, 5, 2) \in \text{fermat}, \\
d[y] &= 4 \in \text{even}.
\end{align*}
\]
A variation of \(\mathcal{P}\) is \(\mathcal{P}' = \langle \{\text{fermat}(x, y, z, 2), \text{even}(y)\}; \ x, y, z \in \{1, \ldots, 10\}\rangle\). We find
\[
\text{Sol}(\mathcal{P}') = \{(3, 4, 5), (6, 8, 10), (8, 6, 10)\}
\]
as the solution set of \(\mathcal{P}'\).

From now on, we use the interval expression \([a..b]\) with integers \(a, b\) to denote the integer set \(\{e \mid a \leq e \leq b\}\).
2.1.3 Solving CSPs by Search and Propagation

Equivalence

We view the search for solutions to a CSP as a process of transforming CSPs. This makes it necessary to relate CSPs to each other disregarding their representation. Instead, we use their solution sets, and accordingly define a notion of equivalence.

A natural definition for two CSPs on the same variables is to say that they are equivalent if they have exactly the same solutions. We extend this notion to sets of CSPs.

2.1.3. Definition. Assume a CSP $\mathcal{P}$ and CSPs $\mathcal{Q}_i$, $i \in [1..m]$, that are all on the same sequence of variable. If

$$\text{Sol}(\mathcal{P}) = \bigcup_{i \in [1..m]} \text{Sol}(\mathcal{Q}_i)$$

then we say that $\mathcal{P}$ is equivalent to the union of $\mathcal{Q}_1, \ldots, \mathcal{Q}_k$. $\square$

2.1.4. Example. The CSP $\mathcal{P} = (\mathcal{C}; X; \mathcal{D})$ is equivalent to the union of

$$\mathcal{Q}_1 = (\mathcal{C} \cup \{\text{even}(x)\}; X; \mathcal{D}),$$
$$\mathcal{Q}_2 = (\mathcal{C} \cup \{\text{odd}(x), y \leq 10\}; X; \mathcal{D}),$$
$$\mathcal{Q}_3 = (\mathcal{C} \cup \{\text{odd}(x), y > 10\}; X; \mathcal{D}),$$

where $x, y \in X$ and odd has the expected meaning. $\square$

Solving Algorithm Schema

We are now in a position to give a basic algorithm schema for finding solutions of CSPs using depth-first search; see Fig. 2.1. Three procedures are used. Solve is the main control procedure, Split generates two sub-CSPs from a given CSP, and Propagate performs constraint propagation. Both Split and Propagate maintain equivalence. This means that Solve is correct and complete in the sense that it returns successfully with a solution if and only if one exists — if the computation terminates.

Here are some instances of the Split procedure. We consider the simple case of splitting the CSP $\mathcal{P}$ into just two subproblems $\mathcal{P}_1$ and $\mathcal{P}_2$.

- **Domain splitting** is a commonly used method. $\mathcal{P}, \mathcal{P}_1, \mathcal{P}_2$ differ only in the domain of some variable $x$. In $\mathcal{P}$, it is $D_x$.
  - **Domain partitioning**: $D_x = D_{x,1} \cup D_{x,2}$ is a partitioning. In $\mathcal{P}_1$ the domain of $x$ is $D_{x,1}$ while in $\mathcal{P}_2$ it is $D_{x,2}$.
  - **Enumeration** is a special case: we have $D_{x,1} = \{e\}$ and $D_{x,2} = D_x - \{e\}$, for some value $e \in D_x$. 


**Chapter 2. Rule-Based Constraint Programming**

**Solve** : CSP $\mathcal{P} \rightarrow \langle$ solution sol, Boolean success$\rangle$

$\mathcal{P} := \text{Propagate}(\mathcal{P})$

if solution sol detected in $\mathcal{P}$ then
  return $\langle$ sol, true$\rangle$
else if inconsistency detected in $\mathcal{P}$ then
  return $\langle\emptyset, \text{false}\rangle$
else
  $\mathcal{PS} := \text{Split}(\mathcal{P})$
  repeat
    choose and remove $\mathcal{P}$ from $\mathcal{PS}$
    $\langle$ sol, success$\rangle := \text{Solve}(\mathcal{P})$
  until success or $\mathcal{PS} = \emptyset$
  return $\langle$ sol, success$\rangle$
end

**Split** : CSP $\mathcal{P} \rightarrow \langle$ CSPs $\mathcal{P}_1, \ldots, \mathcal{P}_n$$\rangle$

// $\mathcal{P}$ is equivalent to the union of $\mathcal{P}_1, \ldots, \mathcal{P}_n$, $n \geq 2$

**Propagate** : CSP $\mathcal{P} \rightarrow$ CSP $\mathcal{P'}$

// $\mathcal{P'}$ is equivalent to $\mathcal{P}$ but possibly “simpler”

Figure 2.1: CSP solver schema with depth-first search and propagation

- **Constraint splitting**: Some constraints have an obvious disjunctive form, which affords splitting. Suppose $|x| = y$ occurs in $\mathcal{P}$. We can obtain the CSPs $\mathcal{P}_1, \mathcal{P}_2$ by replacing $|x| = y$ in $\mathcal{P}$ with $x = y$ and $-x = y$, resp.

The choices that are made when splitting generally have great influence on the solving performance. Consequently, heuristics have been developed to guide the splitting process. In the case of enumeration, one needs to decide which variable to enumerate, and in which order the domain values are tried. The general first-fail heuristic often performs well: it chooses the variable that is most constrained, e.g., has a smallest domain [Haralick and Elliott, 1980]. Specialised heuristics are used in many application domains.

### 2.1.4 Constraint Propagation and Local Consistency

The **Propagate** procedure of Fig. 2.1 is the most interesting one for us. Constraint propagation aims at transforming the CSP into an equivalent one that can be solved easier. Constraint propagation can thus be viewed as deduction,
and the task is to control it in such a way that the computational cost is low but the inferred knowledge – in the form of constraints – is useful for subsequent propagation or search.

Constraint propagation is generally characterised by the properties of the resulting CSP. **Local consistency** notions are used for this purpose. The term ‘local’ reflects the observation that constraint propagation usually does not establish global consistency.

The most important local consistency notions in this work are generalised arc-consistency, bounds-consistency, and path-consistency. We define them now.

**Generalised Arc-Consistency**

Constraint propagation takes place for one constraint at a time in this case. The aim is to obtain the 'smallest possible' variable domains; in other words, to derive all variable-value disequality constraints. (Throughout this work, we use the term “disequality” for ≠ and “inequality” for >, ≥, ≤ and <.)

**2.1.5. Definition.** The constraint $C(x_1, \ldots, x_n)$ is **generalised arc-consistent (GAC)** if

$$\text{for all } x_i, i \in [1..n], \text{ and all } e \in D_i \text{ we have } e \in C[x_i].$$

Recall that $C[x_i]$ stands for \{ $d[x_i] \mid d \in C$ \}. A CSP is generalised arc-consistent if each of its constraints is [Mohr and Masini, 1988].

In short, every domain value must participate in a local solution. The atomic step leading to GAC is: if there is some $e \in D_i$ for some $x_i$ such that $e \notin C[x_i]$, then the domain is reduced by $D_i := D_i - \{e\}$.

**2.1.6. Example.** Consider the CSP

$$\mathcal{P} = \langle \{x^2 + y^2 = z^2, \text{even}(y)\}; x, y, z \in [1..10] \rangle.$$  

$\mathcal{P}$ is not generalised arc-consistent, since for the variable $y$ and $9 \in D_y$ we find $9 \notin \text{even}[y]$. GAC-enforcing propagation of the constraint even must thus infer the constraint $y \neq 9$, or equivalently reduce the domain of $y$ by $D_y := D_y - \{9\}$.

In fact, propagating the constraint even($y$) leads to $y \in \{2, 4, 6, 8, 10\}$.

Complete GAC-enforcing constraint propagation in $\mathcal{P}$ results in

$$\mathcal{P}' = \langle \{x^2 + y^2 = z^2, \text{even}(y)\}; x \in \{3, 6, 8\}, y \in \{4, 6, 8\}, z \in \{5, 10\} \rangle,$$

which is equivalent to $\mathcal{P}$ but makes the solutions more explicit.
GAC on Conjunctive Constraints. Here is a fact about generalised arc-consistency that is useful in several of the following chapters.

2.1.7. Lemma. Consider two constraints $C_1, C_2$ that share at most one variable. If $C_1$ and $C_2$ are generalised arc-consistent individually, then the conjunctive constraint $C = C_1 \land C_2$ is generalised arc-consistent.

Proof. Suppose $C_1(X_1, y)$ and $C_2(X_2, y)$, sharing only the variable $y$, are generalised arc-consistent. Clearly, for each value in $D_y$ there is a solution $d_1$ of $C_1$ and $d_2$ of $C_2$, respectively. We can form a solution $d$ of $C$ by just requiring $d[X_1] = d_1$ and $d[X_2] = d_2$, since $X_1$ and $X_2$ are disjoint. So every value in $D_y$ can be extended to a solution of $C$. The remaining cases are straightforward. □

Generalised arc-consistency is a strong local consistency notion. It may also be computationally expensive: in general, the cost of establishing it on a constraint is exponential in the arity of the constraint. Sometimes computationally cheaper constraint propagation toward a weaker local consistency is more useful.

Bounds-Consistency

The following local consistency notion just checks the bounds of a domain, instead of every contained value.

2.1.8. Definition. Assume that $C(x_1, \ldots, x_n)$ is a constraint such that each of its variables $x_i$ has a totally ordered domain in which $\min(D_i)$ and $\max(D_i)$ are defined accordingly. $C$ is **bounds-consistent (BC)** if

$$\text{for all } x_i \text{ we have } \min(D_i) \in C[x_i] \text{ and } \max(D_i) \in C[x_i].$$

2.1.9. Example. Consider again

$$\mathcal{P} = \langle \{x^2 + y^2 = z^2, \text{even}(y)\}; x, y, z \in [1..10] \rangle.$$

The equivalent CSP

$$\mathcal{P}' = \langle \{x^2 + y^2 = z^2, \text{even}(y)\}; x \in \{3..8\}, y \in \{4..8\}, z \in \{5..10\} \rangle,$$

is bounds-consistent. □

Bounds-consistency is entailed by generalised arc-consistency, but is usually cheaper to establish. A significant representational benefit of using BC instead of GAC is that interval domains remain intervals: establishing BC cannot result in 'holes' in the domains. Intervals require little space to be represented, in contrast to unrestricted sets.
2.2. Rule-Based Programming

Path-Consistency

Finally, we introduce a local consistency notion considering multiple constraints at a time.

2.1.10. Definition. A CSP of only binary constraints is **path-consistent** \((PC)\) [Montanari, 1974] if for every triple of variables \(x, y, z\) we have

\[
C(x, z) = \{ (a, c) \mid \text{there exists } (a, b) \in C(x, y) \text{ and } (b, c) \in C(y, z) \}.
\]

It is assumed here that a unique constraint \(C(u, w)\) for each pair of variables \(u, w\) exists, and that \(C(u, w) = C^{-1}(w, u)\). By \(C^{-1}\) we mean the inverse relation of the binary relation \(C\).

In contrast to the cases of generalised arc-consistency and bounds-consistency, establishing path-consistency may require modification of the constraints, while the variable domains remain the same.

2.1.11. Example. Consider a simple graph-colouring problem. The corner points of a rectangle are to be coloured in red or green; connected corners must have different colours. In our CSP formulation we have a variable for each vertex, ranging over the colours, and disequality constraints for connected vertices:

\[
\begin{align*}
\mathcal{P} &= \langle \mathcal{C}; v_1, \ldots, v_4 \in D \rangle, \\
D &= \{\text{red, green}\}, \\
\mathcal{C} &= \{v_1 \neq v_2, v_2 \neq v_3, v_3 \neq v_4, v_4 \neq v_1\}.
\end{align*}
\]

We kept implicit the true constraints between the two unconnected vertices, namely \(\langle D^2, \langle v_1, v_3 \rangle \rangle\) and \(\langle D^2, \langle v_2, v_4 \rangle \rangle\). \(\mathcal{P}\) is not path-consistent. Replacing \(\mathcal{C}\) in \(\mathcal{P}\) by

\[
\mathcal{C}' = \mathcal{C} \cup \{v_1 = v_3, v_2 = v_4\},
\]

we obtain a CSP \(\mathcal{P}'\) that is equivalent to \(\mathcal{P}\) and path-consistent.

Path-consistency plays a central role in the view on qualitative temporal and spatial reasoning in which binary qualitative relations are represented as constraints.

2.2 Rule-Based Programming

When a program is a set of rules and the computation process consists of a repeated application of the rules, we speak of rule-based programming. By a rule we understand a premise–conclusion pair.
Rules can be found in various places in computer science. Automata in theoretical computer science are based on transition functions, which we can view as sets of rules in the form $s_1, a \rightarrow s_2$ where $s_1, s_2$ are states and $a$ is an input symbol. Reasoning systems in computational logic are quite naturally rule-based, being based on logical calculi that consist of rules over logic formulas. In the field of term rewriting, rules are used to implement directed equational reasoning.

As a concrete programming paradigm, the rule-based approach received much attention in the 1970s with the rise of production systems in Artificial Intelligence. A production rule operates on the elements of the working memory of a production system and describes how they are changed. This development led to the general-purpose language OPS5 [Forgy, 1981] used for programming expert systems.

Logic Programming is a second rule-based formalism from that time (see e.g. [Lloyd, 1987]). It is realised in the Prolog language. A rule (clause) in a logic program relates an atomic formula in the conclusion (the clause head) with a sequence of atomic formulas in the premise (the clause body). The rules in a program are used to prove a goal, and compute a result in the form of a substitution in the process.

Interestingly, term unification, which is at the core of logic programming systems, is itself amenable to a rule-based view. The Martelli-Montanari unification algorithm comprises six rules that can be used to decide whether a set of term equations has a unifying substitution. In the affirmative case, the algorithm yields a most general such substitution [Martelli and Montanari, 1982].

It is instructive to note the classification of rule-based systems with respect to forward chaining and backward chaining approaches. In a forward chaining system, of which production systems are an instance, inference adds derived information to simplify the problem. In a backward chaining system, such as Prolog, the reasoning starts from the goal and, via the rules, attempts to find facts supporting a proof. The rules embody a case distinction, and backtracking is used to explore the cases.

### 2.2.1 Rule-Based Constraint Programming

Constraint logic programming (CLP) originated from logic programming in the 1980s [Jaffar and Maher, 1994]. Hence, also CLP languages as such are rule-based. The observation that the unification operation in logic programming is just a special case of constraint solving led to the CLP(X) scheme [Jaffar and Lassez, 1987], in which X represents the domain of constraint solving (term equalities, arithmetic constraints over reals, finite domain constraints, etc.). In this light, the Martelli-Montanari unification algorithm which solves term equations is the first rule-based constraint solving method.

Another path to rule-based constraint programming originated in the area of term rewriting. The language ELAN [Borovanský et al., 1998] implements an
2.2. Rule-Based Programming

approach to computation and deduction based on conditional rewrite rules and controlled by strategies. Its application to constraint programming is described in [Kirchner and Ringeissen, 1998] and [Castro, 1998].

The hybrid language CLAIRE integrates rules and search into an imperative (resp. object-oriented) language [Caseau et al., 2002]. Its use of logic rules enables declarative programming of the type useful for constraint propagation.

In this context, we also mention [Apt, 1998] in which an account of constraint programming from a proof-theoretic perspective is given. Two classes of proof rules are distinguished, 'deterministic' rules formalising constraint propagation, and 'splitting' rules, which correspond precisely to the procedures Propagate and Split, resp., of the general CSP solution algorithm in Fig. 2.1. In this proof-theoretic view, solving a CSP is regarded as proving it from its solutions.

**Concurrent constraint programming** (CCP) situates the interaction and synchronisation of agents in constraint logic programming [Saraswat, 1993]. Agents use *Ask* and *Tell* operations to publish and query partial information in the form of constraints on shared variables. The constraints are managed in the *constraint store*, which is a set of constraints. AKL [Carlson et al., 1995] and subsequently the Oz language [Smolka, 1995] and the associated Mozart system embody this approach.

Another realisation of the CCP paradigm is the *Constraint Handling Rules* (CHR) language. CHR is a declarative high-level language specifically designed for rule-based constraint programming [Frühwirth, 1998]. It is implemented as a language extension that is compiled to the underlying host language; implementations exist in different systems, including the Prolog-based SICStus [M. Carlsson et al., 2004] and Java [Abdennadher et al., 2002]. CHR uses a committed-choice, forward-chaining approach and is intended for constraint propagation. It relies on the host language to provide the search mechanism needed for full constraint solving. In the logic programming approach to constraint programming, CHR is the language of choice to write constraint solvers.

As CHR is closest to our view of rule-based constraint programming, we present it in some detail. CHR supports two principal types of rules:

- **propagation rules**: $H_1, \ldots, H_k \Rightarrow G_1, \ldots, G_l \mid B_1, \ldots, B_m$,
- **and simplification rules**: $H_1, \ldots, H_k \Leftrightarrow G_1, \ldots, G_l \mid B_1, \ldots, B_m$.

All atomic rule elements can be viewed as constraints, but a distinction is made between defined and primitive constraints:

- the atoms $H_1, \ldots, H_k$ ($k \geq 1$) of the rule head are defined constraints,
- the atoms $G_1, \ldots, G_l$ ($l \geq 0$) of the rule guard are built-in constraints,
- the atoms $B_1, \ldots, B_m$ ($m \geq 1$) of the rule body are arbitrary constraints.
Built-in constraints are provided by the host language (and can also be procedure calls). Defined constraints are managed by the CHR runtime system in the CHR constraint store. Their definition is, in fact, given by the rules.

A CHR rule is executed by first matching its head atoms against constraints in the constraint store. If a match is found, the guard atoms are tested. In case of success, the body atoms are imposed as constraints. If the rule is a simplification rule, additionally the head atoms are removed from the CHR constraint store. This process is repeated until no rule matches with successful guards.

The availability of simplification rules makes CHR very expressive. Propagation rules just add implied constraints to the constraint store, while simplification rules facilitate non-monotonic updating, so the constraint store can be freely managed. This, and the additionally available host language, makes CHR very suitable for high-level design and prototyping of constraint propagation algorithms.

An important issue entailed by the non-monotonicity of simplification rules is that the user must pay attention to confluence and termination of the induced rewriting system.

### 2.2.2 Rule-Based Constraint Propagation

We now formally introduce our notion of constraint propagation rule.

#### 2.2.1. DEFINITION. Assume that $A$ and $B$ are sequences of constraints such that the constraints in $A$ and $B$ are on the variables $X$ with domains $D$. The expression

$$A \rightarrow B$$

is a **constraint propagation rule**. We call $A$ the *condition* and $B$ the *body* of the rule. Rules act as functions on CSPs. The application of a rule to a CSP with the variables $X$ is given by

$$(A \rightarrow B)(\langle C, X, D \rangle) := \begin{cases} 
(C \cup B, X, D) & \text{if } A \subseteq C, \\
(C, X, D) & \text{otherwise}.
\end{cases}$$

The rule $A \rightarrow B$ is *correct* if

$$\text{Sol}(\langle A, X, D \rangle) \subseteq \text{Sol}(\langle B, X, D \rangle).$$

In words, a solution of $A$ is always one of $B$ as well. 

We capture constraint propagation by such rules. It is easy to verify that the application of a correct rule to a CSP yields an equivalent CSP. The aim is to find *useful* rules, that is, those whose body makes the solutions of a CSP more explicit.
2.2. Example.

\[ x < y, \quad y < z \implies x < z \]

is a constraint propagation rule. Since all solutions of \( \{x < y, y < z\}, (x, y, z), D \) are solutions of \( \{x < z\}, (x, y, z), D \) as well, the rule is correct. □

Local Consistency by Rules

The result of constraint propagation is typically characterised by the established local consistency. We now take the inverse view and give simple rule-based characterisations of some local consistency notions, notably generalised arc-consistency and bounds-consistency.

Let us from now on understand \( y \neq a \) with the variable \( y \) and the constant \( a \) as a unary constraint and equally as the domain reduction operation \( D_y := D_y \setminus \{a\} \). That is, we assume that node consistency is maintained. Node consistency [Mackworth, 1977] is the local consistency requiring for a unary constraint \( (C_R, x) \) with \( x \in D_x \) that \( C_R = D_x \), whereas generally we only have \( C_R \subseteq D_x \).

2.2.3. FACT. Suppose \( C \) is a constraint on \( X = x_1, \ldots, x_n \).

- Generalised arc-consistency (Def. 2.1.5) on the constraint \( C \) is established if all correct rules of the form

\[ C(X) \implies x_i \neq e \]

are applied exhaustively. A rule of this form is correct exactly if \( e \not\in C[x_i] \).

- Bounds-consistency (Def. 2.1.8) on the constraint \( C \) is established if all correct rules of the form

\[ C(X) \implies x_i \neq e \quad \text{where} \quad e \in \{\min(D_{x_i}), \max(D_{x_i})\} \]

are applied exhaustively. A rule of this form is correct exactly if \( e \not\in C[x_i] \).

Alternatively we can formulate that bounds-consistency on \( C \) is established if all correct rules of the form

\[ C(X) \implies x_i < e \quad \text{or} \quad C(X) \implies x_i > e \]

are applied exhaustively. □

It is not difficult to see how these characterisations follow directly from the respective definitions.

It is instructive to point out here how constraint propagation can be viewed in terms of the constraint language used in the rules: GAC is obtained by stating all variable-value disequalities, while BC is enforced by variable-value inequalities. [Maher, 2002b] studies this topic in detail.
Membership Rules

A specific class of constraint propagation rules are the **membership rules**, introduced in [Apt and Monfroy, 2001]. These propagation rules have the form

\[ C(x_1, \ldots, x_n, y_1, \ldots, y_m), \ x_1 \in S_1, \ldots, x_n \in S_n \rightarrow y_1 \neq a_1, \ldots, y_m \neq a_m, \]

where each \( S_i \) is a set of constants, and each \( a_i \) is a constant. An expression \( x_i \in S_i \) is a unary constraint on \( x_i \), but in the presence of node consistency it can also be viewed as the simple check \( D_i \subseteq S_i \) on the current domain of \( x_i \). We require \( S_i \neq \emptyset \) for all \( i \in [1..n] \). We also assume that the variables \( x_1, \ldots, x_n, y_1, \ldots, y_m \) are pairwise distinct. We call \( C \) the constraint associated with the rule.

In the following, the constraint associated with a rule is usually clear from the context or irrelevant; we then omit it from the notation. If an \( S_i \) is equal to the variable base domain, then the (always satisfied) condition \( x_i \in S_i \) is often omitted as well. When each set \( S_i \) in a membership rule is a singleton set, we call the rule an **equality rule**.

**2.2.4. Example.** Consider the constraint \( C = \{(0,0),(0,1),(1,1)\} \) on the variables \( x, y \) with the base domain \( D = \{0,1\} \). The rules

\[
\begin{align*}
y \in \{0\} & \rightarrow x \neq 1, \\
x \in \{1\} & \rightarrow y \neq 0,
\end{align*}
\]

associated with \( C \), are correct. \( \square \)

The relevance of membership rules for constraint satisfaction problems with finite domains stems from the following observations [Apt and Monfroy, 2001]:

- constraint propagation can be achieved naturally by repeated application of membership rules;
- in particular, the notion of generalised arc-consistency can be characterised in terms of membership rules;
- for constraints explicitly defined on small finite domains, all correct membership rules can be automatically generated;
- many rules used in specific constraint solvers written in the CHR (Constraint Handling Rules) language are in fact membership rules.

**2.2.5. Example.** Reconsider the constraint and the rules of Example 2.2.4. The two rules establish GAC on their associated constraint. \( \square \)

Membership rules and the template of a GAC-enforcing rule stated in Fact 2.2.3 are clearly connected. Namely, the correctness condition in Fact 2.2.3 is inserted into the condition of the rule. Moreover, the form of this correctness condition changes from a test on the constraint into tests on the variable domains. This has practical benefits since the constraint definition is irrelevant.