Electronic Notes in Theoretical Computer Science 3 (1996) *to appear*

Solving Binary CSP Using Computational Systems

Carlos Castro

INRIA Lorraine & CRIN
615 rue du jardin botanique, BP 101,
54602 Villers-lès-Nancy Cedex, France
e-mail: `Carlos.Castro@loria.fr`

Abstract

In this paper we formalise CSP solving as an inference process. Based on the notion of Computational Systems we associate actions with rewriting rules and control with strategies that establish the order of application of the inferences. The main contribution of this work is to lead the way to the design of a formalism allowing to better understand constraint solving and to apply in the domain of CSP the knowledge already developed in Automated Deduction.

Keywords: Constraint Satisfaction Problems, Computational Systems, Rewriting Logic.

1 Introduction

In the last twenty years many work has been done on solving Constraint Satisfaction Problems, CSP. The solvers used by constraint solving systems can be seen as encapsulated in black boxes. In this work we formalise CSP solving as an inference process. We are interested in description of constraint solving using rule-based algorithms because of the explicit distinction made in this approach between deduction rules and control. We associate actions with rewriting rules and control with strategies which establish the order of application of the inferences. Our first goal is to improve our understanding of the algorithms developed for solving CSP once they are expressed as rewriting rules coordinated by strategies. Extending the domain of application of Rewriting Logic to CSP is another motivation for this work. This leads the way to the design of a formalism allowing to apply the knowledge already developed in the domain of Automated Deduction. To verify our approach we have implemented a prototype which is currently executable in ELAN [13], a system implementing computational systems.

This paper is organised as follows. Section 2 contains some definitions and notations. Section 3 gives a brief overview of CSP solving. Section 4 presents

© *1996 Elsevier Science B. V.*

in details the computational system we have developed. Finally, Section 5 concludes the paper.

2 Basic Definitions and Notation

In this section we formalise CSP. The basic concepts and definitions that we use are based on [10,11,24].

Definition 2.1 [CSP]

An *elementary constraint* $c^?$ is an atomic formula built on a signature $\Sigma = (\mathcal{F}, \mathcal{P})$, where $\mathcal{F}$ is a set of ranked function symbols and $\mathcal{P}$ a set of ranked predicate symbols, and a denumerable set $\mathcal{X}$ of variable symbols. Elementary constraints can be combined with usual first-order connectives and quantifiers. We denote the set of constraints built from Σ and $\mathcal{X}$ by $\mathcal{C}(\Sigma, \mathcal{X})$. Given a structure $\mathcal{D} = (D, I)$, where D is the carrier and I is the interpretation function, a $\langle\Sigma, \mathcal{X}, \mathcal{D}\rangle$-CSP is any set $C = \{c_1^?, \ldots, c_n^?\}$ such that $c_i^? \in \mathcal{C}(\Sigma, \mathcal{X})\ \forall i = 1, \ldots, n$.

We denote a set of constraints C either by $C = \{c_1^?, \ldots, c_n^?\}$ or by $C = (c_1^? \wedge \ldots \wedge c_n^?)$. We also denote by $\mathcal{V}ar(c^?)$ the set of free variables in a constraint $c^?$; these are in fact the variables that the constraint constrains. The *arity* of a constraint is defined as the number of free variables which are involved in the constraint:

$$arity(c^?) = \mathcal{C}ard(\mathcal{V}ar(c^?)).$$

In this way we work with a ranked set of constraints $C = \bigcup_{n \geq 0} C_n$, where C_n is the set of all constraints of arity n.

Definition 2.2 [Interpretation]

Let $\mathcal{D} = (D, I)$ be a Σ-structure and $\mathcal{X}$ a set of variables symbols.

- A *variable assignment* wrt I is a function which assign to each variable in $\mathcal{X}$ an element in D. We will denote a variable assignment wrt I by α, and the set of all such functions by $\alpha_{\mathcal{D}}^{\mathcal{X}}$.
- A *term assignment* wrt I is defined as follows:
  - Each variable assignment is given according to α.
  - Each constant assignment is given according to I.
  - If $t_{1\mathcal{D}}, \ldots, t_{n\mathcal{D}}$ are the term assignment of $t_1, \ldots, t_n$ and $f_{\mathcal{D}}$ is the interpretation of the n-ary function symbol f, then $f_{\mathcal{D}}(t_{1\mathcal{D}}, \ldots, t_{n\mathcal{D}}) \in D$ is the term assignment of $f(t_1, \ldots, t_n)$. We will denote a term assignment wrt I and α by $\alpha(t_{\mathcal{D}})$.
- A formula in $\mathcal{D}$ can be given a truth value, true (**T**) or false (**F**), as follows:
  - If the formula is an atom $p(t_1, \ldots, t_n)$, then the truth value is obtained by calculating the value of $p_{\mathcal{D}}(t_{1\mathcal{D}}, \ldots, t_{n\mathcal{D}})$, where $p_{\mathcal{D}}$ is the mapping assigned to p by I and $t_{1\mathcal{D}}, \ldots, t_{n\mathcal{D}}$ are the term assignments of $t_1, \ldots, t_n$ wrt I.
  - If the formula has the form $(\neg A), (A \wedge B), (A \vee B), (A \rightarrow B)$, or $(A \leftrightarrow B)$, then the truth value of the formula is given by the following table:

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \rightarrow B$	$A \leftrightarrow B$
T	**T**	**F**	**T**	**T**	**T**	**T**
T	**F**	**F**	**F**	**T**	**F**	**F**
F	**T**	**T**	**F**	**T**	**T**	**F**
F	**F**	**T**	**F**	**F**	**T**	**T**

· If the formula has the form $\exists x A$, then the truth value of the formula is true if there exists $d \in D$ such that A has truth value true wrt I and $\alpha_{|x \mapsto d}$, where $\alpha_{|x \mapsto d}$ is α except that x is assigned by d; otherwise, its truth value is false.

· If the formula has the form $\forall x A$, then the truth value of the formula is true if, for all $d \in D$, we have that A has truth value true wrt I and $\alpha_{|x \mapsto d}$; otherwise, its truth value is false.

We denote by $\alpha(A_{\mathcal{D}})$ the interpretation of a formula A wrt I and α.

Definition 2.3 [Satisfiability]

Let Σ be a signature and $\mathcal{D}$ be a Σ-structure:

- Given a formula A and an assignment α, we say that $\mathcal{D}$ *satisfies* A *with* α if

  $$\alpha(A_{\mathcal{D}}) = \mathbf{T}.$$

  This is also denoted by

  $$\mathcal{D} \models \alpha(A).$$

- A formula A is *satisfiable in* $\mathcal{D}$ if there is some assignment α such that

  $$\alpha(A_{\mathcal{D}}) = \mathbf{T}.$$

- A is *satisfiable* if there is some $\mathcal{D}$ in which A is satisfiable.

Definition 2.4 [Solution of CSP]

A solution of $c^?$ is a mapping from $\mathcal{X}$ to D that associates to each variable $x \in \mathcal{X}$ an element in D such that $c^?$ is satisfiable in $\mathcal{D}$. The solution set of $c^?$ is given by:

$$Sol_{\mathcal{D}}(c^?) = \{\alpha \in \alpha_{\mathcal{D}}^{\mathcal{X}} | \alpha(c^?) = \mathbf{T}\}.$$

A solution of C is a mapping such that all constraints $c^? \in C$ are satisfiable in $\mathcal{D}$. The solution set of C is given by:

$$Sol_{\mathcal{D}}(C) = \{\alpha \in \alpha_{\mathcal{D}}^{\mathcal{X}} | \alpha(c_i^?) = \mathbf{T} \;\; \forall i = 1, \ldots, n\}.$$

Finally, in order to carry out the constraint solving process we introduce the following definition:

Definition 2.5 [Membership constraints]

Given a variable $x \in \mathcal{X}$ and a non-empty set $D_x \subseteq D$, the *membership constraint* of x is the relation given by

$$x \in^? D_x.$$

A $\langle \Sigma, \mathcal{X}, \mathcal{D} \rangle$-CSP C' with *membership constraints* is a $\langle \Sigma, \mathcal{X}, \mathcal{D} \rangle$-CSP C where $C' = C \bigcup \{x \in ? D_x\}_{x \in \mathcal{X}}$

We use these membership constraints to make explicit the domain reduction process during the constraint solving. In practice, the sets D_x have to be set up to D at the beginning of the constraint solving process, and during the processing of the constraint network they will be eventually reduced. In the standard literature of constraint solving the term *domain reduction* is generally used to make reference to constraint propagation. Since domains are fixed once the interpretation is chosen, the membership constraints allows to propagate the information in a clear and explicit way. From a theoretical point of view, a membership constraint does not differ from a constraint in the set C; its solution set is defined in the same way.

3 Constraint Solving

In this work we consider CSPs in which the carrier of the structure is a finite set and the constraints are only unary or binary. This class of CSP is known as Binary Finite Constraint Satisfaction Problems or simply Binary CSP [17]. For the graphical representation of this kind of CSP general graphs have been used, that is why CSP are also known as networks of constraints [21]. We associate a graph G to a CSP in the following way. G has a node for each variable $x \in \mathcal{X}$. For each variable $x \in \mathcal{V}ar(c^?)$ such that $c^? \in C_1$, G has a loop, an edge which goes from the node associated to x to itself. For each pair of variables $x, y \in \mathcal{V}ar(c^?)$ such that $c^? \in C_2$, G has two opposite directed arcs between the nodes associated to x and y. The constraint associated to arc (x, y) is similar to the constraint associated to arc (y, x) except that its arguments are interchanged. This representation is based on the fact that the first algorithms to process CSP analyse the values of only one variable when they check a constraint.

Example 3.1 Let $\Sigma = (\{3\}, \{\leq, \neq\})$, where $arity(3) = 0$, $arity(\leq) = arity(\neq) = 2$, $\mathcal{X} = \{x_1, x_2, x_3\}$, $\mathcal{D} = (\{1,2,3,4,5\} \subset \mathbb{N}, \{\leq_\mathcal{D}, \neq_\mathcal{D}\})$, and $3_\mathcal{D}, \leq_\mathcal{D}$ and $\neq_\mathcal{D}$ are interpreted as usual in the natural numbers. Considering the $\langle\Sigma, \mathcal{X}, \mathcal{D}\rangle$-CSP $C = \{x_1 \leq^? 3, x_1 \neq^? x_2, x_1 \neq^? x_3, x_2 \neq^? x_3\}$. If we join the membership constraints $x_1 \in^? D_{x_1}, x_2 \in^? D_{x_2}$ and $x_3 \in^? D_{x_3}$ and these sets D_{x_i} are set up to $D_{x_1} = D_{x_2} = D_{x_3} = \{1,2,3,4,5\}$, the graph which represents this CSP is showed in the Figure 1.

For a given CSP we denote by n the number of variables, by e the number of binary constraints and by a the size of the carrier ($a = \mathcal{C}ard(D)$.) We use $node(G)$ and $arc(G)$ to denote the set of nodes and arcs of graph G, respectively.

Typical tasks defined in connection with CSP are to determine whether a solution exists, and to find one or all the solutions. In this section we present three categories of techniques used in processing CSP: Searching Techniques, Problem Reduction Techniques, and Hybrid Techniques. Kumar's work [14] is an excellent survey on this topic.

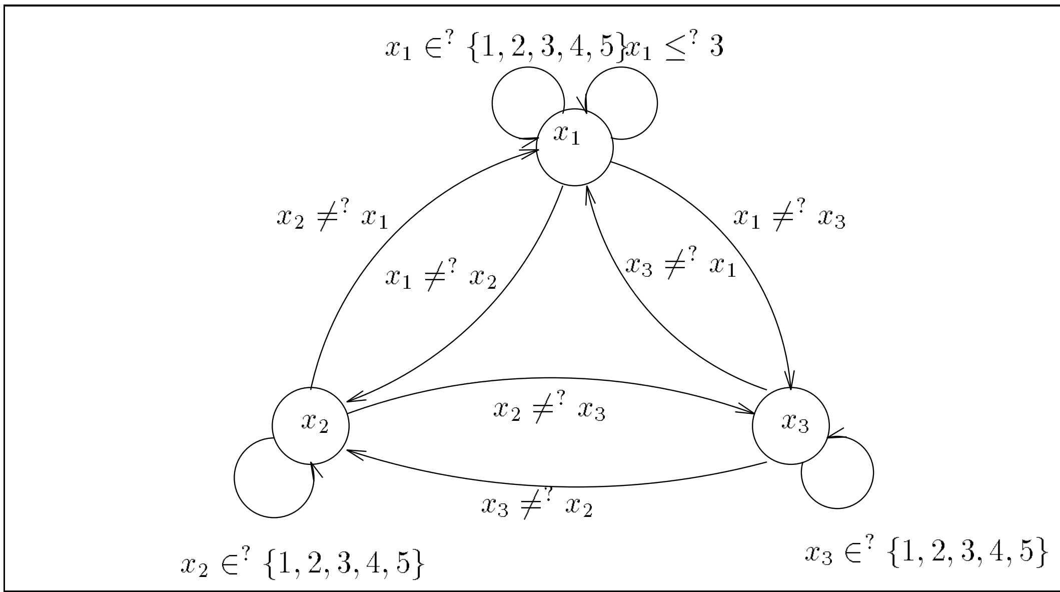

Fig. 1. Graph representation for a Binary CSP

3.1 Searching Techniques in CSP

Searching consists of techniques for systematic exploration of the space of all solutions. The simplest force brute algorithm *generate-and-test*, also called *trial-and-error search*, is based on the idea of testing every possible combination of values to obtain a solution of a CSP. This generate-and-test algorithm is correct but it faces an obvious combinatorial explosion. Intending to avoid that poor performance the basic algorithm commonly used for solving CSPs is the *simple backtracking* search algorithm, also called *standard backtracking* or *depth-first search with chronological backtracking*, which is a general search strategy that has been widely used in problem solving. Although backtracking is much better than generate and test, one almost always can observe pathological behaviour. Bobrow and Raphael have called this class of behaviour *thrashing* [4]. Thrashing can be defined as the repeated exploration of subtrees of the backtrack search tree that differ only in inessential features, such as the assignments to variables irrelevant to the failure of the subtrees. The time complexity of backtracking is $O(a^n e)$, i.e., the time taken to find a solution tends to be exponential in the number of variables [18]. In order to improve the efficiency of this technique, the notion of problem reduction has been developed.

3.2 Problem Reduction in CSP

The time complexity analysis of backtracking algorithm shows that search efficiency could be improved if the possible values that the variables can take is reduced as much as possible [18]. Problem reduction techniques transform a CSP to an equivalent problem by reducing the values that the variables can take. The notion of equivalent problems makes reference to problems which have identical set of solution. Consistency concepts have been defined in order to identify in the search space classes of combinations of values which could

not appear together in any set of values satisfying the set of constraints. Mackworth [17] proposes that these combinations can be eliminated by algorithms which can be viewed as removing inconsistencies in a constraint network representation of the problem and he establishes three levels of consistency: node, arc and path-consistency. These names come from the fact that general graphs have been used to represent this kind of CSP. It is important to realize that the varying forms of consistency algorithms can be seen as *approximation algorithms*, in that they impose *necessary* but not always *sufficient* conditions for the existence of a solution on a CSP.

We now give the standard definitions of node and arc-consistency for a binary network of constraints and we present basic algorithms to achieve them.

3.2.1 Node-Consistency

Definition 3.2 [Node consistency]

Given a variable $x \in \mathcal{X}$ and a unary constraint $c^?(x) \in C$, the node associated to x is *consistent* if

$$\forall \alpha \in \alpha_{\mathcal{D}}^{\mathcal{X}} : \alpha \in Sol_{\mathcal{D}}(x \in^? D_x) \Rightarrow \alpha \in Sol_{\mathcal{D}}(c^?(x)).$$

A network of constraints is *node-consistent* if all its nodes are consistent.

Figure 2 presents the algorithm NC-1 which is based on Mackworth [17]. We assume that before applying this algorithm, there is an initialisation step that set up to D the set D_x associated to variable x in the membership constraint $x \in^? D_x$. The time complexity of NC-1 is $O(an)$ [18], so node consistency is always established in time linear in the number of variables by the algorithm NC-1.

```
procedure NC-1;
1  begin
2    for each x ∈ X do
3      for each α ∈ Sol_D(x ∈? D_x) do
4        if α(c?(x)) = F then
5          D_x ← D_x\α(x);
6        end_if
7      end_do
8    end_do
9  end
```

Fig. 2. Algorithm NC-1 for node-consistency

3.2.2 Arc-Consistency

Definition 3.3 [Arc consistency]

Given the variables $x_i, x_j \in \mathcal{X}$ and the constraints $c_i^?(x_i), c_j^?(x_j), c_k^?(x_i, x_j) \in C$, the arc associated to $c_k^?(x_i, x_j)$ is *consistent* if

$$\forall \alpha \in \alpha_{\mathcal{D}}^{\mathcal{X}} \ \ \exists \alpha' \in \alpha_{\mathcal{D}}^{\mathcal{X}} : \alpha \in Sol_{\mathcal{D}}(x_i \in^? D_{x_i} \wedge c_i^?(x_i))$$

$\Rightarrow \alpha' \in Sol_{\mathcal{D}}(x_j \in^? D_{x_j} \wedge c_j^?(x_j) \wedge c_k^?(\alpha(x_i), x_j))$.

A network of constraints is *arc-consistent* if all its arcs are consistent.

The first three algorithms developed to achieve arc-consistency are based on the following basic operation first proposed by Fikes [8]: Given two variables x_i and x_j, both of which are node-consistent, and the constraint $c^?(x_i, x_j)$, if $\alpha \in Sol_{\mathcal{D}}(x_i \in^? D_{x_i})$ and there is no $\alpha' \in Sol_{\mathcal{D}}(x_j \in^? D_{x_j} \wedge c^?(\alpha(x_i), x_j))$ then $\alpha(x_i)$ has to be deleted from D_{x_i}. When that has been done for each $\alpha \in Sol_{\mathcal{D}}(x_i \in^? D_{x_i})$ then arc (x_i, x_j) is consistent (but that no means that arc (x_j, x_i) is consistent.) This idea is embodied in the function REVISE of Figure 3. The time complexity of REVISE is $O(a^2)$, quadratic in the size of the variable's domain [18].

function REVISE((x_i, x_j)): **boolean**
1 **begin**
2 RETURN $\leftarrow$ **F** ;
3 **for** each $\alpha \in Sol_{\mathcal{D}}(x_i \in^? D_{x_i})$ **do**
4 **if** $Sol_{\mathcal{D}}(x_j \in^? D_{x_j} \wedge c^?(\alpha(x_i), x_j)) = \emptyset$ **then**
5 $D_{x_i} \leftarrow D_{x_i} \backslash \alpha(x_i)$;
6 RETURN $\leftarrow$ **T** ;
7 **end_if**
8 **end_do**
9 **end**

Fig. 3. Function REVISE

At least one time we have to apply function REVISE to each arc in the graph, but it is obvious that further applications of REVISE to the arcs (x_j, x_k), $\forall x_k \in \mathcal{X}$, could eliminate values in D_{x_j} which are necessary for achieving arc-consistency in the arc (x_i, x_j), so reviewing only once each arc will not be enough. The first three algorithms developed to achieve arc-consistency use the same basic action REVISE but they differ in the strategy they apply REVISE.

Algorithm AC-1

AC-1 reviews, applying REVISE, each arc in an iteration. If at least one set D_x is changed all arcs will be reviewed. This process is repeated until no changes ocurr in all sets. Figure 4 presents the simplest algorithm to achieve arc-consistency, where Q is the set of binary constraints to be reviewed.

The worst case complexity of AC-1 is $O(a^3ne)$ [18]. The obvious inefficiency in AC-1 is that a successful revision of an arc on a particular iteration causes all the arcs to be revised on the next iteration whereas in fact only a small fraction of them could possibly be affected.

```
procedure AC-1;
1  begin
2    Q ← {(x_i, x_j) | (x_i, x_j) ∈ arcs(G), x_i ≠ x_j};
3    repeat
4      change ← false ;
5      for each (x_i, x_j) ∈ Q do
6        change ← change or REVISE((x_i, x_j));
7      end_do
8    until ¬change
9  end
```

Fig. 4. Algorithm AC-1 for arc-consistency

Algorithm AC-3

AC-1 can be evidently improved if after the first iteration we only review the arcs which could be affected by the removal of values. This idea was first implemented by Waltz' filtering algorithm [26] and captured later by Mackworth's algorithm AC-2 [17]. The algorithm AC-3 proposed by Mackworth [17] also uses this idea. Figure 5 presents AC-3. If we assume that the constraint graph is connected ($e \geq n - 1$) and time complexity of REVISE is $O(a^2)$, time complexity of AC-3 is $O(a^3 e)$, so arc-consistency can be verified in linear time in the number of constraints [18].

```
procedure AC-3;
1  begin
2    Q ← {(x_i, x_j) | (x_i, x_j) ∈ arcs(G), x_i ≠ x_j};
3    while Q ≠ ∅ do
4      select and delete any arc (x_i, x_j) ∈ Q;
5      if REVISE((x_i, x_j)) then
6        Q ← Q ∪ {(x_k, x_i) | (x_k, x_i) ∈ arcs(G), x_k ≠ x_i, x_k ≠ x_j};
7      end_if
8    end_do
9  end
```

Fig. 5. Algorithm AC-3 for arc-consistency

In [20] Mohr and Henderson propose the algorithm AC-4 whose worst-case time complexity is $O(ea^2)$ and they prove its optimality in terms of time. AC-4 drawbacks are its average time complexity, which is too near the worst-case time complexity, and even more so, its space complexity which is $O(ea^2)$. In problems with many solutions, where constraints are large and arc-consistency removes few values, AC-3 runs often faster than AC-4 despite its non-optimal time complexity [25]. Moreover, in problems with a large number of values in variable domains and with weak constraints, AC-3 is often used instead of AC-4 because of its space complexity. Two algorithms AC-5 have been developed, one by Deville and Van Hentenryck [7] and another by Perlin [23].

They permit exploitation of specific constraint structures, but reduce to AC-3 or AC-4 in the general case. Bessière [1] proposed the algorithm AC-6 which keeps the optimal worst-case time complexity of AC-4 while working out the drawback of space complexity, AC-6 has an $O(ea)$ space complexity. However, the main limitation of AC-6 is its theoretical complexity when used in a search procedure. In [2] Bessière proposes an improved version of AC-6, AC-6+, which uses constraint bidirectionality (a constraint is bidirectional if the combination of values a for a variable x_i and b for a variable x_j is allowed by the constraint between x_i and x_j if and only if b for x_j and a for x_i is allowed by the constraint between x_j and x_i.) This algorithm was improved later by Bessière and Régin with their AC-6++ algorithm [3]; by coincidence in the same workshop Freuder presented his AC-7 algorithm [9]. As our aim in this work is to introduce a new framework to model CSP, we use here only AC-1 and AC-3 algorithms because we need a very simple data structures to implement them.

In general, the complexity analysis of consistency algorithms shows that they can be thought of as a low-order polynomial algorithms for exactly solving a relaxed version of a CSP whose solution set contains the set of solutions to the CSP. The more effort one puts into finding the approximation the smaller the discrepancy between the approximating solution set and the exact solution set.

3.3 Hybrid Techniques

As backtracking suffers from thrashing and consistency algorithms can only eliminate local inconsistencies, hybrid techniques have been developed. In this way we obtain a complete algorithm that can solve all problems and where thrashing has been reduced. Hybrid techniques integrate constraint propagation algorithms into backtracking in the following way: whenever a variable is instantiated, a new CSP is created; a constraint propagation algorithm can be applied to remove local inconsistencies of these new CSPs [27]. Embedding consistency techniques inside backtracking algorithms is called Hybrid Techniques. A lot of research has been done on algorithms that essentially fit the previous format. In particular, Nadel [22] empirically compares the performance of the following algorithms: Generate and Test, Simple Backtracking, Forward Checking, Partial Lookahead, Full Lookahead, and Really Full Lookahead. These algorithms primarily differ in the degrees of arc consistency performed at the nodes of the search tree. These experiments indicate that it is better to apply constraint propagation only in a limited form.

4 A Computational System for Solving Binary CSP

The idea of solving constraint systems using computational systems was firstly proposed by Kirchner, Kirchner and Vittek in [12] where they define the concept of computational systems and describe how a constraint solver for symbolic constraints can be viewed as a computational system aimed at comput-

ing solved forms for a class of considered formulas called constraints. They point out some advantages of describing constraint solving processes as computational systems over constraint solving systems where solvers are encapsulated in black boxes, such as reaching solved forms more efficiently with smart choices of rules, easier termination proofs and possibly partly automated, description of constraint handling in a very abstract way, and easy combination of constraint solving with other computational systems. In this section we briefly present computational systems and then describe in details our system for solving Binary CSP.

4.1 Computational Systems

Following [12], a computational system is given by a signature providing the syntax, a set of conditional rewriting rules describing the deduction mechanism, and a strategy to guide application of rewriting rules. Formally, this is the combination of a rewrite theory in rewriting logic [19], together with a notion of strategy to efficiently compute with given rewriting rules. Computation is exactly application of rewriting rules on a term and strategies describe the intented set of computations, or equivalently in rewriting logic, a subset of proofs terms.

4.2 Solved Forms

Term rewriting repeatedly transforms a term into an equivalent one, using a set of rewriting rules, until a normal form is eventually obtained. The solved form we use is defined with the notion of basic form.

Definition 4.1 [Basic form]

A *basic form* for a CSP P is any conjunction of formula of the form

$$\bigwedge_{i\in I}(x_i \in^? D_{xi}) \wedge \bigwedge_{j\in J}(x_j =^? v_j) \wedge \bigwedge_{k\in K}(c^?(x_k)) \wedge \bigwedge_{l,m\in M}(c^?(x_l, x_m))$$

equivalent to P such that

- $\forall i_1, i_2 \in I, i_1 \neq i_2 \Rightarrow x_{i1} \neq x_{i2}$
- $\forall i \in I, D_{xi} \neq \emptyset$
- $\forall j_1, j_2 \in J, j_1 \neq j_2 \Rightarrow x_{j_1} \neq x_{j_2}$
- $\forall i \in I\ \forall j \in J\ x_i \neq x_j$
- $\forall k \in K\ \ \exists i \in I\ \ \exists j \in J, x_k = x_i\ \vee\ x_k = x_j$
- $\forall l \in M\ \ \exists i \in I\ \ \exists j \in J, x_l = x_i\ \vee\ x_l = x_j$

The constraints in the first, second, third and fourth conjunction are called membership, equality, unary and binary constraints, respectively. For each variable we have associated a membership constraint or an equality constraint, the set associated to each variable in the membership constraints must not be empty, and for each variable appearing in the unary or binary constraints there must be associated a membership constraint or an equality constraint.

Variables which are only involved in equality constraints are called *solved variables* and the others *non-solved variables*.

A CSP P in basic form can be associated with a *basic assignment* obtained by assigning each variable in the equality constraints to the associated value v and each variable x in the membership constraints to any value in the set D_x. In this way we can define several forms depending on the level of consistency we are imposing on the constraint set. So, a CSP P in *unary solved form* is a system in basic form whose set of constraints is node consistent, and a CSP P in *binary solved form* is a system in basic form whose set of constraints is arc consistent.

Definition 4.2 [Solved form]

A *solved form* for a CSP P is a conjunction of formulas in basic form equivalent to P and such that all basic assignments satisfy all constraints. A basic assignment of a CSP P in solved form is called *solution*.

4.3 Rewriting Rules

Figure 6 presents **ConstraintSolving**, a set of rewriting rules for constraint solving in CSP. Some ideas expressed in this set of rules are based on Comon, Dincbas, Jouannaud, and Kirchner's work where they present transformation rules for solving general constraints over finite domains [6].

As we explained in section 3.2.2 the first three algorithms to achieve arc-consistency only differ in the strategy they apply a basic action: REVISE. But, following the main idea of Lee and Leung's Constraint Assimilation Algorithm [15], we can also see the algorithm NC-1 presented in section 3.2.1 as a procedure to coordinate the application of a *domain restriction operation*[1] which removes inconsistent values from the set D_x of the membership constraints. So, we could create only one rewriting rule to implement node and arc-consistency but for clarity reasons we avoid merging both techniques and create the rules **Node-Consistency** and **Arc-Consistency**.

Before applying the algorithm NC-1 we start with the membership constraint $x \in^? D_x$ and the unary constraint $c^?(x)$. After applying NC-1 we obtain a modified membership constraint $x \in^? D'_x$, where D'_x is D_x without the values that satisfy $x \in^? D_x$ but do not satisfy $c^?(x)$. This membership constraint capture all constraint information coming from the original two, their solution sets are the same:

$$Sol_{\mathcal{D}}(x \in^? D'_x) = Sol_{\mathcal{D}}(x \in^? D_x \wedge c^?(x)).$$

This is an inference step where a new constraint can be deduced and the original two be deleted. This key idea is captured by **Node-Consistency**, where $RD(x \in^? D_x, c^?(x))$ stands for the set $D'_x = \{v \in D_x \mid c^?(v)\}$. It is important to note that there is not condition to use this rule because also in case that $c^?(x)$ does not constrain any value already constrained by $x \in^? D_x$

[1] This is the name used by Lee and Leung to denote a general operation REVISE which removes inconsistent values of all variables involved in a n-ary constraint $p(x_1, \ldots, x_n)$.

[Node – Consistency]	$x \in^? D_x \;\wedge\; c^?(x) \;\wedge\; C$
	$\Rightarrow\; x \in^? RD(x \in^? D_x, c^?(x)) \;\wedge\; C$
[Arc – Consistency]	$x_i \in^? D_{x_i} \;\wedge\; x_j \in^? D_{x_j} \;\wedge\; c^?(x_i, x_j) \;\wedge\; C$
	$\Rightarrow\; x_i \in^? RD(x_i \in^? D_{x_i}, x_j \in^? D_{x_j}, c^?(x_i, x_j)) \;\wedge$
	$x_j \in^? D_{x_j} \;\wedge\; c^?(x_i, x_j) \;\wedge\; C$
	if $RD(x_i \in^? D_{x_i}, x_j \in^? D_{x_j}, c^?(x_i, x_j)) \neq D_{x_i}$
[Instantiation]	$x \in^? D_x \;\wedge\; C$
	$\Rightarrow\; x =^? \alpha(x) \;\wedge\; C$
	if $\{\alpha\} = Sol_{\mathcal{D}}(x \in^? D_x)$
[Elimination]	$x =^? v \;\wedge\; C$
	$\Rightarrow\; x =^? v \;\wedge\; C\{x \mapsto v\}$
	if $x \in \mathcal{V}ar(C)$
[Falsity]	$x \in^? \emptyset \;\wedge\; C$
	$\Rightarrow\;$ **F**
[Generate]	$x \in^? D_x \;\wedge\; C$
	$\Rightarrow\; x =^? \alpha(x) \;\wedge\; C$ **or** $x \in^? D_x \backslash \alpha(x) \;\wedge\; C$
	if $\alpha \in Sol_{\mathcal{D}}(x \in^? D_x)$

Fig. 6. **ConstraintSolving**: Rewriting rules for solving Binary CSP

we will not modify the original membership constraint but we can eliminate the constraint $c^?(x)$.

The inference step carried out by arc-consistency algorithms can be seen as an initial state with constraints $x_i \in^? D_{x_i}$, $x_j \in^? D_{x_j}$, and $c^?(x_i, x_j)$ and a final state where $x_i \in^? D_{x_i}$ has been eliminated and a new constraint $x_i \in^? D'_{x_i}$ has been created, where D'_{x_i} corresponds to D_{x_i} without the elements which are not compatible with values in D_{x_j} wrt $c(x_i, x_j)$. This is expressed by the inference rule **Arc-Consistency**, where $RD(x_i \in^? D_{x_i}, x_j \in^? D_{x_j}, c^?(x_i, x_j))$ stands for the set $D'_{x_i} = \{v \in D_{x_i} \mid (\exists w \in D_{x_j}) \;\; c^?(v, w)\}$. In this case we require that $RD(x_i \in^? D_{x_i}, x_j \in^? D_{x_j}, c^?(x_i, x_j)) \neq D_{x_i}$ to really go on.

The rewriting rule **Instantiation** corresponds to the variable instantiation. If there is only one assignment α which satisfies $x \in^? D_x$ then the membership constraint is deleted and a new constraint $x =^? \alpha(x)$ is added. This rule makes explicit the dual meaning of an assignment. Algorithmic languages require two different operators for equality and assignment. In a constraint language, equality is used only as a relational operator, equivalent to the corresponding operator in conventional languages. The constraint solving mechanism "assigns" values to variables by finding values for the variables that make the equality relationships true [16].

Elimination express the fact that once a variable has been instantiated we can propagate its value through all constraints where the variable is involved in. In this way we can reduce the arity of these constraints; unary constraints will become ground formulas whose truth value have to be verified and binary constraints will become unary constraints which are more easily tested. Once we apply **Elimination** the variable involved in this rule will become a solved variable. It is important to note the strong relation between **Instantiation** and **Elimination**. Semantically the constraints $x \in^? \{v\}$ is equivalent to $x =^? v$, but for efficiency reasons the use of **Elimination** allows the simplification of the constraint system avoiding further resolution of the membership constraint and the constraints where the variable in involved in. Advantages of this approach have been pointed out since the early works on mathematical formula manipulation where the concept of *simplification* was introduced. Caviness [5] mentions that simplified expressions usually require less memory, their processing is faster and simpler, and their functional equivalence are easier to identify. However, it is necessary to point out that with this choice we lose some information, particularly in case of incremental constraint solving, because we do not know any more where the variable was involved in.

The rule **Falsity** express the obvious fact of unsatisfiability. If we arrive to $D_x = \emptyset$ in a membership constraint $x \in^? D_x$ the CSP is unsatisfiable.

The rule **Generate** express the simple fact of branching. Starting with the original constraint set we generate two subsets. In one of them we assume an instantiation for any variable involved in the membership constraints; in the other subset we eliminate that value from the set involved in the membership constraint associated to that variable. In this way the solution for the original problem will be in the union of the solutions for the subproblems.

Lemma 4.3 *The set of rules* **ConstraintSolving** *is correct and complete.*

Proof: Correctness of rule **Node-Consistency** is reduced to prove that $Sol_{\mathcal{D}}(x \in^? RD(x \in^? D_x, c^?(x))) \subseteq Sol_{\mathcal{D}}(\alpha(x \in^? D_x \wedge c^?(x)))$. By definition $RD(x \in^? D_x, c^?(x)) = D'_x$ where $D'_x = \{v \in D_x \mid c^?(v)\}$, so evidently all solution of $x \in^? RD(x \in^? D_x, c^?(x))$ is solution of $x \in^? D_x \wedge c^?(x)$. To prove completeness we can follow the same idea. Correctness and completness of rule **Arc-Consistency** can be proved using the same schema as for **Node-Consistency**. The prove for rules **Instantiation**, **Elimination**, and **Falsity** is evident. The right hand side of rule **Generate** is equivalent to $(x =^? \alpha(x) \vee x \in^? D_x \backslash \alpha(x)) \;\wedge\; C$. This expresion is equivalent to $x \in^? D_x \;\wedge\; C$, the left hand side of the rule, so rule **Generate** is correct and complete.

Theorem 4.4 *Starting with a CSP P and applying repeatedly the rules in* **ConstraintSolving** *until no rule applies anymore results in* **F** *iff P has no solution or else it results in a solved form of P.*

Proof: Termination of the set of rules is clear since the application of all rules, except one, strictly reduce the size of the set of constraints. The only exception is rule **Instantiation** that does not reduce the set. This rule eliminates a membership constraint and creates an equality constraint. As

membership constraint are only created at the beginning of the constraint solving, one for each variable, this rule is applied at most n times.

When we start constraint solving we have the system $C \wedge x \in^? D_x;\ \forall x \in \mathcal{X}$. Rule **Node-Consistency** eliminates unary constraints from C. **Arc-Consistency** only modifies the sets D_x. Rule **Instantiation** eliminates membership constraints and creates at most one equality constraint per variable. Rule **Eliminate** eventually deletes unary constraints and transforms binary constraints into unary constraints. **Generate** modifies a domain D_x, or deletes a membership constraint and creates an equality constraint. So, if the problem is satisfiable the application of these rules gives a solved form. If the problem is unsatisfiable, i.e., some domain becomes empty, rule **Falsity** will detect that.

4.4 Strategies

As we have mention there are several heuristics to search for a solution in CSP, starting from the brute force generate and test algorithm until elaborated versions of backtracking. The expressive power of computational systems allows to express these different heuristics through the notion of strategy. In this way, for example, a unary solved form can be obtained by applying [**Node-Consistency** | **Falsity**]*, a binary solved form can be obtained by applying [**Arc-Consistency** | **Falsity**]*, and a solved form can be obtained using the strategy [[**Generate; Elimination**] | **Falsity**]* which implements exhaustive searching [2].

We can integrate constraint propagation and searching in order to get a solved form more efficiently than the force brute approach. Let us define the following strategies for applying rules from **ConstraintSolving**:

- **NodeC** :: **Node-Consistency** [[**Instantiation; Elimination**]|**Falsity**]*
- **ArcC** :: **Arc-Consistency** [[**Instantiation; Elimination**]|**Falsity**]*
- **ConsSol1** :: [**NodeC** | **ArcC**]* [[**Generate; Elimination**]|**Falsity**]*
- **ConsSol2** :: [[**NodeC** | **ArcC**]* **Generate; Elimination**]*

The strategy **ConsSol1** implements a preprocessing which verifies node and arc consistency and then carries out an exhaustive search in the reduced problem. The strategy **ConsSol2** implements an heuristic which, once node and arc consistency have been verified, carries out an enumeration step, then verifies again node and arc consistency and so on. **ConsSol2** is a particular version of *Forward Checking* an heuristic widely used in CSP.

4.5 Implementation

We have implemented a prototype of our system which is currently executable in the system ELAN [13], an interpretor of computational systems [3]. To verify

[2] The symbol * means applying a given rule zero or N times over the constraint system.

[3] ELAN is available via anonymous ftp at `ftp.loria.fr` in the directory `/pub/loria/protheo/softwares/Elan`. Further information can be obtained at

our approach we have implemented constraint solving using two versions of arc consistency: AC-1 and AC-3 [4]. The benchmarks which we have carried out are consistent with the well known theoretical and experimental results in terms of constraint checking, where AC-3 is obviously better than AC-1. Using the non determinism of ELAN we have easily implemented Forward Checking, the most popular hybrid technique. In Appendix A we present an overview of our implementation. All details about this prototype can be obtained at `http://www.loria.fr/~castro/PROJECTS/csp.html`.

5 Conclusion

We have implemented a prototype of a computational system for solving Binary CSP. We have verified how computational systems are an easy and natural way to describe and manipulate Binary CSP. The main contributions of this work can be seen from two points of view. First, we have formalised algorithms to solve Binary CSP in a way which makes explicit difference between actions and control that until now were embeded in black boxes like algorithms. Second, we have extended the domain of application of Rewriting Logic. The distinction between actions and control allows us to better understand the algorithms for constraint solving which we have used. As our aim in this work was only to apply the expressive power of computational systems to better understand constraint propagation in CSP we did not care about efficiency in searching for a solution, so as future work we are interested in efficiency considerations related to our implementation. As a near future work we are interested in the analysis of the data structures which will allow us to implement more efficient versions of arc-consistency algorithms. We hope that powerful strategy languages will allow us to evaluate existing hybrid techniques for constraint solving and design new ones.

Acknowledgement

I am grateful to Dr. Claude Kirchner for his theoretical support and Peter Borovanský and Pierre-Etienne Moreau for their help concerning the implementation.

References

[1] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190, 1994.

`http://www.loria.fr/equipe/protheo.html/PROJECTS/ELAN/elan.html`

[4] The rewriting system presented in this work allows the direct implementation of AC-1. Implementing AC-3 only required to add a rewriting rule to check the constraints which could be affected by the constraint propagation. For simplicity reasons we do not include it here.

[2] C. Bessière. A fast algorithm to establish arc-consistency in constraint networks. Technical Report TR-94-003, LIRMM Université de Montpellier II, January 1994.

[3] C. Bessière and J.-C. Régin. An arc-consistency algorithm optimal in the number of constraint checks. In *Proceedings of the Workshop on Constraint Processing, ECAI'94, Amsterdam, The Netherlands*, pages 9–16, 1994.

[4] D. G. Bobrow and B. Raphael. New Programming Languages for Artificial Intelligence Research. *Computing Surveys*, 6(3):153–174, September 1974.

[5] B. F. Caviness. On Canonical Forms and Simplification. *Journal of the ACM*, 17(2):385–396, April 1970.

[6] H. Comon, M. Dincbas, J.-P. Jouannaud, and C. Kirchner. A Methodological View of Constraint Solving. Working paper, 1996.

[7] Y. Deville and P. V. Hentenryck. An efficient arc consistency algorithm for a class of csp problems. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 325–330, 1991.

[8] R. E. Fikes. REF-ARF: A System for Solving Problems Stated as Procedures. *Artificial Intelligence*, 1:27–120, 1970.

[9] E. C. Freuder. Using metalevel constraint knowledge to reduce constraint checking. In *Proceedings of the Workshop on Constraint Processing, ECAI'94, Amsterdam, The Netherlands*, pages 27–33, 1994.

[10] J. H. Gallier. *Logic for Computer Sciences, Foundations of Automatic Theorem Proving*. Harper and Row, 1986.

[11] H. Kirchner. On the Use of Constraints in Automated Deduction. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 128–146. Springer-Verlag, 1995.

[12] C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In P. V. Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers*, pages 131–158. The MIT press, 1995.

[13] C. Kirchner, H. Kirchner, and M. Vittek. *ELAN, User Manual*. INRIA Loraine & CRIN, Campus scientifique, 615, rue du Jardin Botanique, BP-101, 54602 Villers-lès-Nancy Cedex, France, November 1995.

[14] V. Kumar. Algorithms for Constraint-Satisfaction Problems: A Survey. *Artificial Intelligence Magazine*, 13(1):32–44, Spring 1992.

[15] J. H. M. Lee and H. F. Leung. Incremental Querying in the Concurrent CLP Language IFD-Constraint Pandora. In K. M. George, J. H. Carroll, D. Oppenheim, and J. Hightower, editors, *Proceedings of the 11th Anual Symposium on Applied Computing, SAC'96, Philadelphia, Pennsylvania, USA*, pages 387–392, February 1996.

[16] W. Leler. *Constraint Programming Languages, Their Specification and Generation*. Addison-Wesley Publishing Company, 1988.

[17] A. K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.

[18] A. K. Mackworth and E. C. Freuder. The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. *Artificial Intelligence*, 25:65–74, 1985.

[19] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[20] R. Mohr and T. C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.

[21] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.

[22] B. Nadel. Tree Search and Arc Consistency in Constraint-Satisfaction Algorithms. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, pages 287–342. Springer-Verlag, 1988.

[23] M. Perlin. Arc consistency for factorable relations. *Artificial Intelligence*, 53:329–342, 1992.

[24] P. van Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT press, 1989.

[25] R. J. Wallace. Why AC-3 is almost always better than AC-4 for establishing arc concsistency in CSPs. In *Proceedings IJCAI-93*, pages 239–245, 1993.

[26] D. Waltz. Understanding lines drawings of scenes with shadows. In P. H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.

[27] M. Zahn and W. Hower. Backtracking along with constraint processing and their time complexities. *Journal of Experimental and Theoretical Artificial Intelligence*, 8:63–74, 1996.

A Implementation

In ELAN, a logic can be expressed by its syntax and its inference rules. The syntax of the logic can be described using mixfix operators. The inference rules of the logic are described by conditional rewrite rules. The language provides three levels of programmation:

- First the design of a logic is done by the so-called *super-user*. In our case that is a description in a generic way of the constraint solving process.
- The logic can be used by the *programmer* in order to write a specification.
- Finally, the *end-user* can evaluate queries valid in the specification, following the semantics described by the logic.

In our implementation the top level of the logic description is given by the *super-user* in the module presented in figure A.1.

LPL *Solver_ CSP_ Int* **description**
specification description
  part *Variables* **of sort** list[identifier]
  part *Values* **of sort** list[int]
end
  query of sort list[*formule*]
  result of sort *csp*
  modules *Solver_ CSP[Variables*,int,*Values]*
  start with (*Solved_ Form*) *CreateCSP*(**query**)
end of LPL description

Fig. A.1. Logic description

This module specifies that the *programmer* has to provide a specification module which has to include two parts: *Variables* and *Values*. As an example we can consider the specification module presented in figure A.2.

specification *My_ variables_ and_ values*
Variables
  X1 X2
Values
  1 2 3 4 5
end of specification

Fig. A.2. End-user specification

The sorts *list, identifier* and *int* are built-in, and the *query sort* and *result sort* are defined by the super-user. Sort list[*formule*] defines the data structure of the query, in this case, a list of constraints. The sort *csp* is a data structure consisting of three list; the first one records the membership constraints, the second one records the equality constraints, and the third one records the unary and binary constraints. Once the *programmer* has defined the logic, and has provided a query term, ELAN will process in the following way. The symbol *CreateCSP* will apply on the query term, then using the strategy *Solved_ Form*, included in the module *Solver_ CSP*, ELAN will iterate until no rule applies anymore. *CreateCSP* uses the constructors *CreateLMC*, to create the list of membership constraints, and *CreateC*, to create the list of unary and binary constraints from the list of de formula L given by the *end-user*[5].

The strategy *Solved_ Form* control the application of the rules as is showed in the figure A.3. This strategy implements local consistency with exhaustive search. If we eliminate the sub-strategy **dont know choose**(*Generate*) we obtain a particular version of AC-1 algorithm.

Finally, in figure A.4 we present rule *Node-Consistency*. This rule applies the strategy *Strategy_ Node-Consistency*, presented in figura A.5, on a *csp*

[5] Creation of the list of unary and binary constraints is not only a copy of the list L, because for each binary constraint $c^?(x_i, x_j)$ we have to create its inverse $c^?(x_j, x_i)$.

```
strategy Solved_Form
repeat
  dont care choose (
      dont care choose (Node-Consistency)
      ||
      dont care choose (Arc-Consistency)
      ||
      dont care choose (Instantiation)
      ||
      dont care choose (Elimination)
      ||
      dont care choose (Falsity)
      ||
      dont know choose (Generate)
      )
endrepeat
end of strategy
```

Fig. A.3. Strategy *Solved_ Form*

with at least one element in the list of unary and binary constraints. Strategy *Strategy_ Node-Consistency* uses rule *GetUnaryConstraint* to get the first unary constraint in the list of unary and binary constraints. If there exists a unary constraint the strategy will apply rule *Node-Consistency_ 1*, if the variable involved in the unary constraint is in the list of membership constraints, or rule *Node-Consistency_ 2*, if the variable is in the list of equality constraints. In the set **ConstraintSolving** we use only one rule to verify node consistency, but we have implemented two versions sligtly differents. This is an implementation choice, as we have a list for the membership constraints and another one for the equality constraints, it is easy to profite this information. The same explanation is valid for arc consistency, where we have created four rules to implement the general version presented in the set **ConstraintSolving**.

```
rules for csp
declare
    x : var;
    v : Type;
    D : list[Type];
    c : formule;
    C,lmc,lec : list[formule];
    P : csp;
bodies
[Node-Consistency] CSP(lmc,lec,c.C) => P
          where P:= (Strategy_Node-Consistency)CSP(lmc,lec,c.C)
end
[Node-Consistency_1] CSP(x in? D.lmc,lec,c.C)
          => CSP(x in? ReviseDxWRTc(x,D,c).lmc,lec,C)
end
[Node-Consistency_2] CSP(lmc,x =? v.lec,c.C) =>
          CSP(lmc,x =? v.lec,C)
          if SatisfyUnaryConstraint(x,v,c)
...
end
```

Fig. A.4. Rules to implement Node Consistency

```
strategy Strategy_Node-Consistency
    dont care choose (GetUnaryConstraint)
    dont care choose (
          dont care choose (GetVarOfUnaryConstraintInLMC)
          dont care choose (Node-Consistency_1)
          ||
          dont care choose (GetVarOfUnaryConstraintInLEC)
          dont care choose (Node-Consistency_2)
          )
end of strategy
```

Fig. A.5. Strategies to implement Node Consistency