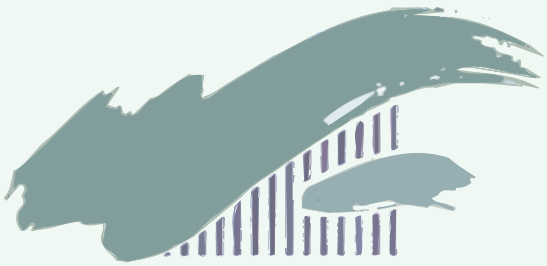

UNIVERSITE D'ORLEANS
Faculté des Sciences
LIFO

Laboratoire d'Informatique Fondamentale d'Orléans
4, rue Léonard de Vinci, BP 6759
F-45067 Orléans Cedex 2
FRANCE

Rapport de Recherche

www : http://www.univ-orleans.fr/SCIENCES/LIFO/

A Declarative View of CSP Consistencies

Gérard Ferrand, Arnaud Lallouet
Université d'Orléans, LIFO

Rapport N^o **2004-10**

A Declarative View of CSP Consistencies

Gérard Ferrand and Arnaud Lallouet

Université d'Orléans — LIFO
BP 6759 — F-45067 Orléans cedex 2
{Gerard.Ferrand | Arnaud.Lallouet}@lifo.univ-orleans.fr

Abstract. We provide here a model-theoretic characterization of the approximations computed by a consistency while solving a finite domain CSP. At first, we propose a definite CLP program which allows to characterize the computed approximation by its fixpoints. This framework is general enough to be applied to (hyper)arc-consistency, bound-consistency or any approximation scheme. Moreover, motivated by other approaches of CSP representations by logic programs with stable model semantics, the computed approximation is also represented in a natural way by the stable models of another (normal) CLP program.

Keywords : Constraint Logic Programming; CSP; Consistencies; Stable models.

1 Introduction

This paper presents a link between Logic Programming and Constraint Programming on finite domains. For a Constraint Satisfaction Problem (or CSP), we propose a logic program under stable model semantics which represents declaratively not only the solutions of the problem but also all the intermediate computation steps called *consistent states*.

Constraint Satisfaction Problems are a powerful knowledge representation technique which combine the declarativity of a high level specification with the existence of powerful resolution techniques. Emerging in the mid-70's, AI techniques for solving CSPs, namely *consistency* maintenance, have been since then highly successful on classical Operation Research problems, especially on finite domains. These techniques essentially consist in the computation of an over-appproximation of the CSP's solutions by so-called *constraint propagation*, a process by which forbidden values communicate via the variables shared by constraints to remove new forbidden values across the CSP. But as the problem size and the complexity of the solving algorithms increase, there is a need for an in-depth understanding of this process. For example, chaotic iterations [9, 1] is a high-level formalization of the operational semantics of the solvers. In this approach, a solver is modeled by a set of constraint reduction operators and their associated consistency by their greatest common fixed-point.

The contribution of this paper is the representation of the approximation computed by a consistency by a declarative semantics of a CLP program. But since a consistency is an over-approximation, it cannot be the standard least fixpoint semantics, and its computation can be done by a top-down goal-directed procedure such as CSLD-resolution. The long-term motivation of this work is twofold: firstly, we want to get an in-depth understanding of constraint propagation in order to reason on constraint programming systems and this logical formalization can be one of the tools which can be used for this purpose. Secondly, it can help to establish a deeper connexion between CSPs and the stable model semantics of Logic Programs, in order to improve cross-fertilization of two domains which share a large part of implementation techniques.

We first propose to transform a CSP into a definite CLP program P and show that the greatest fixpoint of its associated operator T_P coincides with the consistency's computed approximation. But since the solving process does not end with the first consistency enforcement, we also show that consistent states obtained after arbitrary labeling steps are also modeled by downward closures of T_P starting from suitably restricted interpretations. A second CLP program P_{neg} is obtained by separating in the negative and positive parts the atoms which are certainly deleted from the ones which may belong to a solution. This allows to model the individual contribution of each propagation rule. But since the program has negations, the semantics we use is the stable model semantics [11]. We prove that consistent states are stable models of the program.

It follows that this work is not about the semantics of CLP but focusses on CSPs for which we extend the logical semantics to consistent states and not only to solutions as usual in declarative problem modeling. Its purpose is to model current (and also future) consistencies but not to propose new consistencies for CSPs. This would be a fairly different work which involves an experimental evaluation of the incidence of the new consistency to the resolution of some class of problems. In contrast with the usual ways of describing solvers, namely with Constraint Handling Rules (CHRs) [10] or indexicals [23], we provide a model-theoretic framework instead of a description of transformations.

In summary, these programs both propose a system of rules with a logical semantics to model a CSP's computed approximation. The paper is organized as follows: in section 2, preliminaries about the CLP scheme and CSP solving are given, section 3 introduces a common framework for both of them, section 4 is devoted to the study of the first (definite) program and its semantics, section 5 gives the normal program P_{neg} and studies its stable models, section 6 describes an implementation of the transformation from CSP to logic program.

2 Preliminaries

2.1 Semantics of (Constraint) Logic Programs

In CLP(D), term unification is extended with constraint solving on a domain D (pure LP is viewed as CLP(*Herbrand*) with the constraint $=$). It needs a language to express the constraints which are integrated into Logic Programming [12]. The constraint language is classically defined on a set of variables V and a signature containing a set Σ of function symbols and a set Π_C of constraint predicate symbols. A primitive constraint is an atomic formula $c(t_1, \ldots t_n)$ where $c \in \Pi_C$ and the t_i's are (Σ, V)-terms. A constraint is a particular case of first-order (Π_C, Σ, V)-formula. The pre-interpretation of the constraints is given by a (Π_C, Σ)-structure $\mathcal{D}$ on the domain D.

Let Π_P be a set of program predicates symbols. An atom is an atomic (Π_P, Σ, V)-formula. The interpretation of a predicate p is given by a set of pseudo-atom called D-atom $p(d_1, \ldots, d_n)$ where the d_i's are elements of D. The D-base is the set of all D-atoms and we denote it by $\mathcal{B}$. A $\mathcal{D}$-interpretation is identified to an arbitrary subset of the D-base. A valuation is a mapping from V to D.

A definite clause is a formula $h \leftarrow c, b_1, \ldots, b_n$ where h and the b_i's are atoms and c is a constraint. A $\mathcal{D}$-instance of such a clause is a rule obtained by instanciation by a valuation satisfying the constraint c. A definite program is a set of definite clauses and is identified to the set of $\mathcal{D}$-instances of its clauses. To a given program P is associated an immediate consequence operator $T_P : \mathcal{P}(\mathcal{B}) \rightarrow \mathcal{P}(\mathcal{B})$ defined by $T_P(I) = \{h \mid \exists h \leftarrow B \in P, B \subseteq I\}$. This operator is monotonic and continuous and its least fixed point (in short lfp) is the least $\mathcal{D}$-model of P [12].

Following [12], we call definition of a predicate p the formula

$$\forall X_1, \ldots, X_n \quad p(X_1, \ldots, X_n) \leftrightarrow B_1 \vee \ldots \vee B_m$$

where for all clause $p(t_1, \ldots, t_n) \leftarrow L_1, \ldots, L_k$ defining p in P, B_i is the formula

$$\exists Y_1, \ldots Y_j (\bigwedge_{i=1..n} X_i = t_i \ \wedge \ \bigwedge_{i=1..k} L_i)$$

where $Y_1, \ldots Y_j$ are the variables of the clause. The completion of P, denoted by P^*, is the set of the definitions of P's predicates. The greatest fixed point (in short gfp) of T_P is also the greatest $\mathcal{D}$-model of P^*.

A literal is an atom a or the negation of an atom $\neg a$. If I is an interpretation, then $\neg I$ denotes the set $\{\neg a \mid a \in I\}$ and $\overline{I} = \mathcal{B} - I$. A normal clause is a clause whose body is composed of literals instead of atoms. From this follows the notion of normal program. Various semantics for normal programs have been considered. We are particularily interested by the stable model semantics [11].

The stable model semantics [11] is based on the Gelfond-Lifschitz (or GL-) transformation of the program P with respect to a set of atoms $I \subseteq \mathcal{B}$. Let P/I be the (definite) program obtained by removing from P any clause containing a literal $\neg a$ with $a \in I$ and by removing all negative literals from the body of the remaining clauses. The interpretation I is a stable model of P if I is the least model of P/I. In this case, I can be viewed as a consistent set of beliefs concerning the program P. In general, a program has more than one stable model, each of them being interesting. In [15], this is presented as a novel — very declarative — way of programming called *Stable Logic Programming*. A program is a collection of formulas whose purpose is to specify (or constrain) some class of models. For example, to have the atom g in every stable model with no perturbation of the semantics of g already defined by other clauses, we shall write the clause $f \leftarrow \neg g, \neg f$ where f is a new predicate symbol not appearing in P (and not by adding the fact $g \leftarrow$ which could interfere with other predicates in some stable models). This is usually written in this framework as the shorter integrity constraint $\leftarrow \neg g$ [15, 17]. Stable models have strong connexions with another well-known semantics, the well-founded semantics [22] since the well-founded semantics is the $\subseteq$-least 3-valued stable model [21]. While stable models are defined for general logic program, their implementation needs a grounding of the program (the most widely used being the one of SMODELS [18]) which restricts predicate domains to finite domains. This is also another evidence of the connexion between both fields.

2.2 CSPs

For a family $a = (a_i)_{i \in I}$ and $J \subseteq I$, we denote by $a|_J = (a_i)_{i \in J}$ the restriction of a to the set of indices J. We denote by A^I the cartesian power $I \longrightarrow A$ and by $\mathcal{P}(A)$ the powerset of A.

Let V be a denumerable set of variables and D a domain. Note that the use of a many sorted domain is possible in this context but we limit ourselves to a single domain in this paper for clarity.

Definition 1 (Constraint).
A constraint *is a pair* $c = (W, T)$ *where* $W = var(c) \subseteq V$ *is the* arity *of* c *and* $T = sol(c) \subseteq D^{var(c)}$ *is a table containing the tuples solution of* c.

A *constraint satisfaction problem*, or in short *CSP*, is a set C of constraints. Finite domain constraint solving classically involves a finite domain D, as well as a finite number of constraints (and so of variables). A *solution* is a tuple of D^V satisfying all constraints, i.e. $s \in sol(C) \iff \forall c \in C, s|_{var(c)} \in sol(c)$. Solving a CSP means

finding one (or all, but this is tackled by the same methods) of its solutions. It could be done only by enumeration but the combination of search and approximation methods called *consistencies* yields far better performance in practice.

In this paper, we use the most common approximation, the domain reduction scheme [24, 6, 2] in which variable domains are represented, totally or at least partially by intervals. The approximation consists in pruning *inconsistent* values from variable domains, i.e. those which are not supported by any constraint, according to the values still present in the domains of the constraint's other variables. The best known consistency is the so-called *arc-consistency* [14] which holds for a binary constraint, say $c(X, Y)$, if, for each value d in the domain of X, there is a corresponding value e in the domain of Y such that $c(d, e)$ holds (and conversely by exchanging the role of X and Y). This notion is extended to hyperarc-consistency for n-ary constraints in a straightforward manner: let $c(X_1, \ldots, X_n)$ be a constraint; a value a_1 in the domain of X_1 is consistent if there exists $n-1$ values $a_2, \ldots, a_n$ in the domains of $X_2, \ldots, X_n$ such that $(a_1, \ldots, a_n) \in sol(c)$. A whole CSP is arc-consistent if its variable domains are arc-consistent for all constraints. Bound-consistency is a weaker property in which only the bounds of the variables domains are supposed to be consistent. Variable domains are often considered as new unary constraints whose solutions only are reduced:

Definition 2 (Search state).
A search state *is a family* $(\mathcal{E}_X)_{X \in V}$ *where* $\forall X \in V, \mathcal{E}_X \subseteq D$.

We call *search space* the set $\mathcal{P}(D)^V$ of all search states. Each $\mathcal{E}_X$ represents the current domain of the variable X and the approximation which is defined by a search state is the cartesian product of this family of domains. A solution is found when every variable domain has been reduced to a singleton. We call such a state a *singletonic* search state. Conversely, if a variable domain becomes empty, then the CSP has no solution.

All known consistencies can be represented by the common closure of a set R of *reduction* operators, i.e. monotonic, correct and contracting operators [16, 4, 1]. For the arc-consistency with respect to $c(X, Y)$, it can be formalized by the mapping $r(\mathcal{E}) = \{d \in \mathcal{E}_X \mid \exists e \in \mathcal{E}_Y, c(d, e)\}$ and the one obtained by exchanging the role of X and Y. The set of operators associated to these mappings completely defines the expected consistency, which is obtained by computing their greatest common fixed-point. This can be done by defining an aggregate operator $T(\mathcal{E}) = \bigcap_{t \in R} t(\mathcal{E})$ and iterating it, or by a *chaotic iteration* [9, 1], where only one operator of R is applied at each time, fairness of the scheduling ensuring the confluence. Correctness of the operators is required for not missing any solution. However, completeness, which is not required in general, has to hold for singleton sets. It is simply needed for the solver to discriminate solutions. This fact is of particular importance because a solver replaces the original constraints by the set of operators.

In this context, two extensions can be considered. Firstly, the current solutions of constraints of arbitrary arity can be reduced. This scheme is called the constraint reduction scheme [16, 1], of which the domain reduction scheme is a particular case. Secondly, arbitrary operators can be used, provided that they are correct, monotonic and contractant. All known consistencies can be modeled by such operators and also they can always be defined by a set of rules. Possible rules systems include indexicals [23] and CHRs [10]. Also in [13], we proposed a set of *propagation* rules which express deletion of values in function of previous deletions. In this paper, the rules are intended to be positive, or in other terms, they should express directly what is a consistent state, like CHRs or indexicals. We call R-consistency the stability by such an arbitrary set of operators R.

Example 3 (Arc-consistency).
For example, let C be the following CSP composed of two constraints $k_1(X, Y)$ given

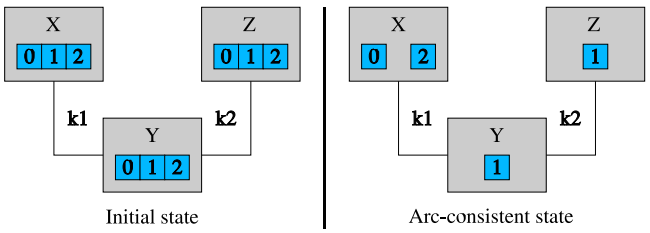

Fig. 1. Arc-consistency

by its solutions $\{(0,1),(1,2),(2,1)\}$ and $k_2(Y,Z)$ defined by $\{(0,2),(1,1)\}$. The original domains of X, Y and Z are $\{0,1,2\}$. Then arc-consistency should remove the value 0 from Y's domain because it has no support in k_1 and this removal implies the removal of 2 from Z's domain because of k_2. The arc-consistent state computed by doing all propagations is depicted in figure 1.

3 A common framework

We shall first proceed towards an integration of the two frameworks. To achieve this, we first express CSPs in the language of CLP. A CSP search state $\mathcal{E}$ is composed of a subset of the domain for each variable. In order to move towards a Logic Programming framework, we put in Π_P a unary *program* predicate symbol x (also denoted by $pred(X)$) for each variable X — and not a constraint predicate, which would remain at the constraint level —. With this bijection $x \leftrightarrow X$, we can identify the search state $\mathcal{E} = (\mathcal{E}_X)_{X \in V} \in \mathcal{P}(D)^V$ with the D-interpretation $\{x(d) \mid d \in \mathcal{E}_X\}$. For example, with two variables X and Y, if $\mathcal{E}$ is the search state $(X \mapsto \{0,1\}, Y \mapsto \{1,2\})$, then this family of domains is identified to the D-interpretation $\{x(0), x(1), y(1), y(2)\}$. We denote usually such an interpretation by $\mathcal{E}$ or I. Let $\mathcal{A}$ be the set of atoms corresponding to variables, i.e. $\{x(d) \mid X \in V, x = pred(X), d \in D\}$. Since we will add other predicate symbols in the next sections, the set $\mathcal{A}$ will be the part of the $\mathcal{D}$-base which corresponds to variables.

Since search states and interpretations are now related, we define what it means for an interpretation to be R-consistent.

Definition 4 (Support rule).
A support rule *is a ground definite rule $h \leftarrow B$ where $h \in \mathcal{A}$ and $B \subseteq \mathcal{A}$.*

It expresses that a value d of X's domain is supported by the other atoms in B.

Example 5 (Support rule, example 3 running).
Let k_1 and k_2 be the constraints defined in example 3. The only support rule for $x(1)$ is `x(1) <- y(2)`. On the other side $y(1)$ has two sets of support rules: `y(1) <- x(0)` and `y(1) <- x(2)` which come from k_1 and `y(1) <- z(1)` which comes from k_2.

Support rules are low-level because they are always instantiated. Hence other kinds of rules are usually preferred to describe consistencies like, for example, indexicals [23] or CHRs [10]. The rules we introduce hereafter, called *arc-consistency rules* can easily be expressed by indexicals or CHRs.

Definition 6 (Arc-consistency rule).
An arc-consistency rule *for a constraint c is a pair $r = (c, X)$ with $X \in var(c)$ whose semantics is:*

$$x(X) \leftarrow c \wedge \bigwedge_{Y \in var(c) - \{X\}} y(Y)$$

Example 7 (Arc-consistency rules, example 5 running).
In our running example, the arc-consistency rules for Y are `y(Y) <- k1(X,Y), x(X)` and `y(Y) <- k2(Y,Z), z(Z)`.

Example 8 (Indexicals).
Consider the constraint $X = Y + 3$. In the indexical language, an arc-consistency rule for X is written `X in dom(Y)+3`. A local consistency rule expressing bound-consistency for X is written `X in min(Y)+3 .. max(Y)+3`.

Arc-consistency rules are not the only kind of rules which can be considered. For bound-consistency, the idea is to express a relaxed constraint involving only the variables' bounds:

Example 9 (Bound-consistency rules).
For the constraint $X = Y$, we can write the following rule to express bound-consistency for X:

```
x(X) <- y(Y1), y(Y2), X >= Y1, X <= Y2.
```

Equality is relaxed into "belonging to the same interval". In the indexical language, this is written `X in min(Y) .. max(Y).`

One can imagine that other kind of consistencies could be expressed by rules in some language (see [5] for a framework for approximate consistencies). In the general case, we call an uninstantiated rule a *local consistency rule*, bound- and arc-consistency rules being two particular cases. Actually, depending on the language, writing an expression for bound- or arc-consistency may be easy or cumbersome, but to a local consistency rule, it can always be associated a set of support rules. In fact, we consider local consistency rules as being a convenient *shortcut* to describe sets of support rules. Of course arc- and bound-consistency rules are special kinds of local consistency rules. Let R be a set of local consistency rules.

Example 10 (Hyperarc-consistency).
The set of rules which achieves hyperarc-consistency is $AC = \{(c, X) \mid c \in C, X \in var(c)\}$. The consistency obtained with these rules coincides with arc-consistency for binary CSPs and with node-consistency for unary constraints. It is possible to define a weaker consistency by considering in R a subset of all possible local consistency rules (in order to model directional arc-consistency [7] for example).

Let $r \in R$ be a local consistency rule. We define the following notations:

- $var(r)$ denotes the predicate of the target variable of the rule (the one which appears in the head, like x in an indexical `X in r`). We assume that var is surjective, that is every variable $X \in V$ has at least a rule $r \in R$ such that $pred(X) = var(r)$.
- $srules(r)$ is the set of support rules $x(d) \leftarrow B$ such that $x = var(r), d \in \mathcal{D}$ and $B \subseteq \mathcal{A}$ which are instance of the local consistency rule r.
- $R(X) = \{r \in R \mid pred(X) = var(r)\}$ is the set of all local consistency rules whose head is x. We assume that $\forall X \in V, R(X) \neq \emptyset$.

Example 11 (srules, example 7 running).
The arc-consistency rule $r_1 = (k_1, X)$ is used to denote the following set of support rules obtained by instanciation:

$$srules(r_1) = \{x(0) \leftarrow y(1), \quad x(1) \leftarrow y(2), \quad x(2) \leftarrow y(1)\}$$

Definition 12 (r-supported atom).
Let $x(d) \in \mathcal{A}$, $r \in R$ and $\mathcal{E} \subseteq \mathcal{A}$. The atom $x(d)$ is r-supported *by $\mathcal{E}$ if:*

$$\exists B \subseteq \mathcal{E} \quad \textit{such that} \quad x(d) \leftarrow B \in srules(r)$$

Definition 13 (r-covered atom).
The atom $x(d)$ is r-covered *by $\mathcal{E}$ if $x(d)$ is r-supported by $\mathcal{E}$ and $x(d) \in \mathcal{E}$.*

Definition 14 (R-consistent interpretation).
An interpretation $\mathcal{E} \subseteq \mathcal{A}$ is R-consistent *if:*

$$\forall x(d) \in \mathcal{E}, \forall r \in R, x = var(r) \Rightarrow x(d) \text{ is } r\text{-supported by } \mathcal{E}$$

Example 15 (Arc-consistency, example 11 running).
Let k_1 and $k2$ be the constraints defined as in example 3. The set of rules AC is composed of the four following consistency rules $r_1 = (k_1, X), r_2 = (k_1, Y), r_3 = (k_2, Y)$ and $r_4 = (k_2, Z)$. The interpretation $\mathcal{E} = \{x(0), x(1), x(2), y(0), y(1), y(2), z(0), z(1), z(2)\}$ is not R-consistent because at least the atom $y(0)$ is not r_2-supported. However, the interpretation $\mathcal{F} = \{x(0), x(2), y(1), z(1)\}$ is.

The following lemma characterizes R-consistent states:

Lemma 16.
An interpretation $\mathcal{E} \subseteq \mathcal{A}$ is R-consistent iff $\forall X \in V, x = pred(X), \forall d \in \mathcal{D}$,

$$x(d) \in \mathcal{E} \Leftrightarrow \forall r \in R(x = var(r) \Rightarrow x(d) \text{is } r\text{-covered in } \mathcal{E}$$

Proof.

$\Rightarrow$ Assume that $\mathcal{E}$ is R-consistent. We first assume that $\forall r \in R$ such that $x = var(r)$, then $x(d)$ is r-covered in $\mathcal{E}$. Let $r \in R$ be such that $x = var(r)$. Since var is surjective, such a r exists. Then $x(d)$ is actually r-covered in $\mathcal{E}$ and therefore $x(d) \in \mathcal{E}$. The other side is obvious.
$\Leftarrow$ straightforward.

4 A logical formulation of CSP approximations

A local consistency rule is also a CLP($\mathcal{D}$) clause. However, the set of such clauses defining the operators is not the expected program. To an arc-consistency rule $r = (c, X)$ may be associated a clause (the only difference with the rule is that $x(X)$ is now also in the body):

$$x(X) \leftarrow c \wedge \bigwedge_{Y \in var(c)} y(Y)$$

This construction is similar to the clause representation of graphs used in [16], for which it is mentioned that it does not enjoy a logic program semantics. In fact, in order to model consistency, we need to get the intersection of the consequences of the rules, and not their union. This is done by introducing for each arc-consistency rule r a new program predicate symbol of the same name r. This should not be confusing since they are not used in the same context. The D-base $\mathcal{B}$ now becomes $\{x(d) \mid X \in V, x = pred(X), d \in D\} \cup \{r(d) \mid r \in R, d \in D\}$. From now on, we shall use the letter I to denote an arbitrary D-interpretation and the letter $\mathcal{E}$ to denote an interpretation $\subseteq \mathcal{A}$.

In order to get the program P corresponding to the R-consistency, we define:

- if $r \in R$ is any local consistency rule,

$$\text{rule-clause}(r) \quad := \quad \{r(d) \leftarrow x(d), B \mid x(d) \leftarrow B \in srules(r)\}$$

In the particular case of an arc-consistency rule $r = (c, X)$, $srules(r)$ is simply the set of instances of the following rule:

$$\text{rule-clause}(r) \quad := \quad r(VX) \leftarrow c \wedge \bigwedge_{Y \in var(c)} y(VY)$$

– for each variable $X \in V$,

$$\text{var-clause}(X) \quad := \quad x(VX) \leftarrow \bigwedge_{r \in R(X)} r(VX)$$

Definition 17 (CLP program associated to a consistency). *The program P associated to the set of local consistency rules R is the CLP($\mathcal{D}$) program:*

$$\{\textit{rule-clause}(r) \mid r \in R\} \cup \{\textit{var-clause}(X) \mid X \in V\}$$

Intuitively, for $r = (c, X)$, rule-clause(r) collects the values maintained in X's domain according to c, and var-clause(X) allows this value only if maintained by all constraints. Note that since there is no fact (because a constraint involves at least one variable and because $R(X) \neq \emptyset$), the least $\mathcal{D}$-model of P is always empty, and thus of no interest. Let $T_P : \mathcal{P}(\mathcal{B}) \longrightarrow \mathcal{P}(\mathcal{B})$ be the monotonic operator associated to P.

Example 18 (example 15 running).
The program P associated to the CSP C under arc-consistency is:

$$\begin{array}{lcl} rc_1 & := & r_1(X) \leftarrow k_1(X,Y),\ x(X),\ y(Y). \\ rc_2 & := & r_2(Y) \leftarrow k_1(X,Y),\ x(X),\ y(Y). \\ rc_3 & := & r_3(Y) \leftarrow k_2(Y,Z),\ y(Y),\ z(Z). \\ rc_4 & := & r_4(Z) \leftarrow k_2(Y,Z),\ y(Y),\ z(Z). \end{array} \qquad \begin{array}{lcl} vc_1 & := & x(X) \leftarrow r_1(X). \\ vc_2 & := & y(Y) \leftarrow r_2(Y),\ r_3(Y). \\ vc_3 & := & z(Z) \leftarrow r_4(Z). \end{array}$$

In order to simplify the presentation, we identify the program P to the set of its instances.

Theorem 19.
Let $I \subseteq \mathcal{B}$ be an interpretation. Then $I = T_P(I)$ is equivalent to:

1. *$\forall r \in R, x = var(r), \forall d \in \mathcal{D}$, we have:*

$$r(d) \in I \Leftrightarrow x(d) \textit{ is r-covered in } I \cap \mathcal{A}$$

2. *$\forall X \in V, x = pred(X), \forall d \in \mathcal{D}$, we have:*

$$x(d) \in I \Leftrightarrow \forall r \in R(x = var(r) \Rightarrow x(d) \textit{ is r-covered in } I \cap \mathcal{A})$$

Proof.

$\Rightarrow$ Assume that $I = T_P(I)$.

1. let $r \in R$ and $x = var(r)$.

$$\begin{aligned} r(d) \in I &\Leftrightarrow r(d) \in T_P(I) \\ &\Leftrightarrow \exists B \text{ such that } x(d) \leftarrow B \in srules(r) \text{ and } B \cup \{x(d)\} \subseteq I \\ &\Leftrightarrow x(d) \in I \text{ and } \exists B \subseteq I \cap \mathcal{A} \text{ such that } x(d) \leftarrow B \in srules(r) \\ &\Leftrightarrow x(d) \text{ is } r\text{-covered in } I \cap \mathcal{A} \end{aligned}$$

2. let $X \in V$ and $x = pred(X)$.

$$\begin{aligned} x(d) \in I &\Leftrightarrow x(d) \in T_P(I) \\ &\Leftrightarrow \forall r \in R, (x = var(r) \Rightarrow r(d) \in I) \text{ (by var-clause)} \\ &\Leftrightarrow \forall r \in R, (x = var(r) \Rightarrow x(d) \text{ is } r\text{-covered in } I \cap \mathcal{A}) \end{aligned}$$

$\Leftarrow$ Assume that (1) and (2) are holding.

$\subseteq$:

$$x(d) \in I \Leftrightarrow \forall r \in R, (x = var(r) \Rightarrow r(d) \in I) \text{ (by hypothesis 2 followed by 1)}$$
$$\Leftrightarrow x(d) \in T_P(I) \text{ (by var-clause)}$$

$\supseteq$:

$$r(d) \in I \Leftrightarrow x(d) \in I \text{ and } \exists B \subseteq I \cap \mathcal{A} \text{ such that } x(d) \leftarrow B \in srules(r)$$
$$\Leftrightarrow \exists B \text{ such that } x(d) \leftarrow B \in srules(r) \text{ and } B \cup \{x(d)\} \subseteq I$$
$$\Leftrightarrow r(d) \in T_(I)$$

Now we get the main result of this section: there exists a correspondence between the fixpoints of T_P and the consistent states. We just have to take care of the atoms $r(d)$ coming from the rules. On the first side, starting from a fixpoint of T_P, we get a consistent state by forgetting the atoms coming from the rules:

Corollary 20.
$$I = T_P(I) \quad \Rightarrow \quad I \cap \mathcal{A} \textit{ is } R\textit{-consistent}$$

Proof. By lemma 16 and (2) of theorem 19.

On the other side, we have to complete a consistent state with suitable atoms coming from the rules:

Corollary 21.
Let $\mathcal{E} \subseteq \mathcal{A}$ be R-consistent. Let set $I = \mathcal{E} \cup \{r(d) \mid x(d)$ is r-covered in $\mathcal{E}$ and $x = var(r)\}$. Then $I = T_P(I)$.

Proof. Obviously, $I \cap \mathcal{A} = \mathcal{E}$. Hence we can apply (1) of theorem 19 and lemma 16.

The least fixpoint of T_P is empty and thus $\emptyset$ is R-consistent. Its greatest fixpoint corresponds to the usual closure first computed by a consistency algorithm. Moreover, the correspondence between fixpoints and consistent states is bijective:

Theorem 22. *The application $I \mapsto I \cap \mathcal{A}$ is a bijection between the set of* fixpoints *of T_P and the set of R-consistent subsets of $\mathcal{A}$.*

Proof. This application (by corollary 20) is injective (by (1) and (2) of theorem 19) and surjective (by corollary 21).

Example 23 (Downward closure, example 18 running).
Let $\mathcal{D} = \{0, 1, 2\}$. We use the arc-consistency rules defined in example 15. The downward iterations of T_P starting from the full Herbrand base $\mathcal{B}$ are detailed as follows, where the notation $p(a; b)$ stands for $p(a), p(b)$:

- $T \downarrow^0 \mathcal{B} = \mathcal{B}$
- $T \downarrow^1 \mathcal{B} = \mathcal{A} \cup \{r_1(0; 1; 2), r_2(1; 2), r_3(0; 1), r_4(1; 2)\}$
- $T \downarrow^2 \mathcal{B} = \{x(0; 1; 2), y(1), z(1; 2)\} \cup \{r_1(0; 1; 2), r_2(1; 2), r_3(0; 1), r_4(1; 2)\}$
- $T \downarrow^3 \mathcal{B} = \{x(0; 1; 2), y(1), z(1; 2)\} \cup \{r_1(0; 2), r_2(1), r_3(1), r_4(1)\}$
- $T \downarrow^4 \mathcal{B} = \{x(0; 2), y(1), z(1)\} \cup \{r_1(0; 2), r_2(1), r_3(1), r_4(1)\} = \text{gfp}(T_P)$.

We get $\text{gfp}(T_P)) \cap \mathcal{A} = \{x(0), x(2), y(1), z(1)\}$ which is exactly the closure obtained by arc-consistency.

The usual solving process of a CSP does not end as the first closure is found. Usually, a variable is chosen and its domain is partitioned in two (domain splitting) or its values are enumerated (labeling) until a solution is found. We recall that a solution for a CSP is a single tuple satisfying all constraints. In the domain reduction framework, it is represented by a *singletonic* search state or interpretation:

Definition 24 (singletonic interpretation).
An interpretation $\mathcal{E} \subseteq \mathcal{A}$ is singletonic *if for all variable $X \in V$, there exists an unique $x(d) \in \mathcal{E}$.*

Thus, for a singletonic interpretation $\mathcal{E}$, there exists a tuple $s_{\mathcal{E}} : V \to D$ such that $\mathcal{E} = \{x(s_{\mathcal{E}}(X)) \mid X \in V\}$. The solutions of the CSP C are simply these singletonic R-consistent interpretations.

According to fixpoint theory, such closures are not always fixpoints of T_P, they cannot be obtained by iteration, but they can be obtained by iteration of the contractant operator associated to T_P, i.e. T'_P defined by $T'_P(I) = I \cap T_P(I)$. The following example illustrates this point:

Example 25 (Solution, example 23 running).
Starting from the first closure obtained in example 23, we can enumerate the domain of X. This is done by removing for example $x(2)$. The interpretation becomes $\{x(0), y(1), z(1)\} \cup \{r_1(0), r_2(1), r_3(1), r_4(1)\}$ which is a singletonic fixpoint and therefore a solution.

5 Approximations and stable models

The greatest fixed-point semantics of P allows to express the computed approximation. In this section, we enhance this framework by introducing a program with negation which allows to characterize more directly the values removed by consistency and has a more intuitive reading.

Let us transform the program P into P_{neg} in order to isolate the contribution of each rule r. For this, we introduce a new program predicate symbol nx for each variable X. Its meaning is that $nx(d)$ is true if the value d has been removed from the domain of x.

Definition 26 (P_{neg}). *Let P_{neg} be the following CLP($\mathcal{D}$) program:*

$$\begin{array}{ll} x(X) \leftarrow \neg nx(X) & \forall X \in V, x = pred(X) \\ nx(X) \leftarrow \neg r(X) & \forall r \in R(X), x = pred(X) \\ \textit{rule-clause}(r) & \forall r \in R \end{array}$$

The intuitive meaning of this program is closer to the idea of consistency: if a value d for a variable X is unsupported by some constraint c, then this value has to be removed. It follows from P_{neg} that $nx(d)$ is true and $x(d)$ false. Because of the semantics of programs with negation, the semantics of this program deserves a study. The stable model semantics expresses nicely the useful notion of "false if the negation is proved" by default.

Example 27 (P_{neg}, example 25 running).
The program P_{neg} associated to the CSP C and arc-consistency is the following:

$$\begin{array}{l} x(X) \leftarrow \neg\ nx(X). \\ y(Y) \leftarrow \neg\ ny(Y). \\ z(Z) \leftarrow \neg\ nz(Z). \\ \\ nx(X) \leftarrow \neg\ r_1(X). \\ ny(Y) \leftarrow \neg\ r_2(Y). \\ ny(Y) \leftarrow \neg\ r_3(Y). \\ nz(Z) \leftarrow \neg\ r_4(Z). \end{array} \qquad \begin{array}{l} r_1(X) \leftarrow k_1(X,Y),\ x(X),\ y(Y). \\ r_2(Y) \leftarrow k_1(X,Y),\ x(X),\ y(Y). \\ r_3(Y) \leftarrow k_2(Y,Z),\ y(Y),\ z(Z). \\ r_4(Z) \leftarrow k_2(Y,Z),\ y(Y),\ z(Z). \end{array}$$

Now the Herbrand base becomes $\mathcal{B}_{neg} = \mathcal{B} \cup \{nx(d) \mid X \in V, d \in D\}$.

Theorem 28.
Let $I \subseteq \mathcal{B}_{neg}$ be an interpretation. Then I is a stable model of P_{neg} is equivalent to:

1. $\forall r \in R, x = var(r), \forall d \in \mathcal{D}$, *we have:*

$$r(d) \in I \Leftrightarrow x(d) \textit{ is r-covered in } I \cap \mathcal{A}$$

2. $\forall X \in V, x = pred(X), \forall d \in \mathcal{D}$, *we have:*

$$nx(d) \in I \Leftrightarrow \exists r \in R\,(x = var(r) \textit{ and } x(d) \textit{ is not r-covered in } I \cap \mathcal{D})$$

3. $\forall X \in V, x = pred(X), \forall d \in \mathcal{D}$, *we have:*

$$x(d) \in I \Leftrightarrow \forall r \in R\,(x = var(r) \Rightarrow x(d) \textit{ is r-covered in } I \cap \mathcal{D})$$

Proof.
For an interpretation I, the program P_{neg}/I obtained by the Gelfond-Lifschitz transformation is the following:

$$\begin{cases} x(d) \leftarrow & \text{if } nx(d) \notin I \\ nx(d) \leftarrow & \text{if } \exists r \in R \text{ such that } x = var(r) \text{ and } r(d) \notin I \\ r(d) \leftarrow B \cup \{x(d)\} & \forall r \in R, x = var(r), \forall d \in \mathcal{D} \text{ such that } x(d) \leftarrow B \in srules(r) \end{cases}$$

We denote by $M(P)$ the least $\mathcal{D}$-model of a definite program P.

$\Rightarrow$ Assume that I is a stable model of P_{neg}, hence $I = M(P_{neg}/I)$.

1. let $r \in R$ and $x = var(r)$.

$$\begin{aligned} r(d) \in I &\Leftrightarrow r(d) \in M(P_{neg}/I) \\ &\Leftrightarrow \exists B \text{ such that } x(d) \leftarrow B \in srules(r) \text{ and } B \cup \{x(d)\} \subseteq M(P_{neg}/I) \\ &\Leftrightarrow \exists B \text{ such that } x(d) \leftarrow B \in srules(r) \text{ and } B \cup \{x(d)\} \subseteq I \\ &\Leftrightarrow x(d) \in I \cap calA \text{ and } \exists B \subseteq I \cap \mathcal{A} \text{ such that } x(d) \leftarrow B \in srules(r) \\ &\Leftrightarrow x(d) \text{ is } r\text{-covered in } I \cap \mathcal{A} \end{aligned}$$

2. let $X \in V$ and $x = pred(X)$.

$$\begin{aligned} nx(d) \in I &\Leftrightarrow nx(d) \in M(P_{neg}/I) \\ &\Leftrightarrow nx(d) \text{ is a fact of } P_{neg}/I \\ &\Leftrightarrow \exists r \in R, (x = var(r) \text{ and } r(d) \notin I) \\ &\Leftrightarrow \exists r \in R, (x = var(r) \text{ and } x(d) \text{ is not } r\text{-covered in } I \cap \mathcal{A}) \end{aligned}$$

3. let $X \in V$ and $x = pred(X)$.

$$\begin{aligned} x(d) \in I &\Leftrightarrow x(d) \in M(P_{neg}/I) \\ &\Leftrightarrow x(d) \text{ is a fact of } P_{neg}/I \\ &\Leftrightarrow nx(d) \notin I \\ &\Leftrightarrow \forall r \in R, (x \neq var(r) \text{ or } x(d) \text{ is } r\text{-covered in } I \cap \mathcal{A}) \\ &\Leftrightarrow \forall r \in R, (x = var(r) \Rightarrow x(d) \text{ is } r\text{-covered in } I \cap \mathcal{A}) \end{aligned}$$

$\Leftarrow$ Assume that (1), (2) and (3) are holding.

- x atoms:

$$\begin{aligned} x(d) \in I &\Leftrightarrow nx(d) \notin I \\ &\Leftrightarrow x(d) \text{ is a fact of } P_{neg}/I \\ &\Leftrightarrow x(d) \in M(P_{neg}/I) \end{aligned}$$

- nx atoms:

$$\begin{aligned} nx(d) \in I &\Leftrightarrow \exists r \in R \text{ such that } x = var(r) \text{ and } r(d) \notin I \\ &\Leftrightarrow nx(d) \text{ is a fact of } P_{neg}/I \\ &\Leftrightarrow nx(d) \in M(P_{neg}/I) \end{aligned}$$

- r atoms: let $r \in R$ and $x = var(r)$.

$$\begin{aligned} r(d) \in I &\Leftrightarrow x(d) \in I \cap \mathcal{A} \text{ and } \exists B \subseteq I \cap \mathcal{A} \text{ such that } x(d) \leftarrow B \in srules(r) \\ &\Leftrightarrow \exists B \text{ such that } x(d) \leftarrow B \in srules(r) \text{ and } B \cup \{x(d)\} \subseteq I \cap \mathcal{A} \\ &\Leftrightarrow \exists B \text{ such that } x(d) \leftarrow B \in srules(r) \text{ and } B \cup \{x(d)\} \subseteq M(P_{neg}/I) \\ &\Leftrightarrow r(d) \in M(P_{neg}/I) \end{aligned}$$

Now we can state the main results: there exists a correspondence between the stable models of P_{neg} and the consistent states. On the first side, starting from a stable model of P_{neg}, we get a consistent state by forgetting the atoms coming from the rules (r-atoms) and from the negative part (nx-atoms):

Corollary 29.
Let I be an interpretation.

$$I \text{ is a stable model of } P_{neg} \Rightarrow I \cap \mathcal{A} \text{ is R-consistent}$$

Proof. By lemma 16 and (3) of theorem 28.

On the other side, we have to complete a consistent state with suitable atoms coming from the rules and negations:

Corollary 30.
Let $\mathcal{E} \subseteq \mathcal{A}$ be R-consistent. Let set $I = \mathcal{E} \cup \{nx(d) \mid x(d) \notin \mathcal{E}\} \cup \{r(d) \mid x(d)$ is r-covered in $\mathcal{E}$ and $x = var(r)\}$. Then I is a stable model of P_{neg}.

Proof. Obviously, $I \cap \mathcal{A} = \mathcal{E}$. Hence we can apply (1) and (2) of theorem 28 and lemma 16.

Moreover, the correspondence between stable models and consistent states is bijective:

Theorem 31. *The application $I \mapsto I \cap \mathcal{A}$ is a bijection between the set of* stable models *of P_{neg} and the set of R-consistent subsets of $\mathcal{A}$.*

Proof. This application (by corollary 29) is injective (by (1), (2) and (3) of theorem 28) and surjective (by corollary 30).

The transformation from P to P_{neg} is quite simple, taking its inspiration from De Morgan laws. Also, P_{neg} has several properties, namely call-consistency (see [3], def. 4.5) and its stable models coincide with the Herbrand models of its completed (see [3], cor. 6.19).

From a logical point of view, the definite program P allows to express the consistency as the greatest $\mathcal{D}$-model of P. But the stable models allows to express more explicitly that the approximation consists in deleting values from the variable domains. For a literal l, being in a stable model M means either to be guessed as true or false, or to be proved, either in a positive or negative way. This is meaningful since:

- $nx(d)$ is proved means that $\neg\, x(d)$ is proved, which holds if there exists a rule $r \in R(X)$ such that $\neg\, r(d)$ is proved, or in other words, if the value d has been rejected from X's domain because of r.

– $\neg\ r(d)$ is proved means that for all solution s of c, such that $d = s(X)$, there exists $Y \in var(c)$ such that $\neg\ y(s(Y))$ is proved. This reflects exactly constraint propagation.

Another difference is that the normal program reflects the way CSPs are compositional. Adding a new constraint to the CSP is modeled by adding new clauses (rule-clause) defining this constraint to the normal program. Clauses defining variables domains and their negative counterparts are unchanged. Unfortunately, the composition is not that natural for the definite program because adding a new constraint implies a modification of the body of var-clause for the variables concerned in order to add a reference to the newly added predicate.

There is a considerable recent work in the area of stable models in order to model CSPs either directly or from a knowledge representation point of view [18, 15, 17, 19, 20]. For example, [17] proposes a translation from a CSPs to a normal logic program whose stable models are the CSP's solutions. Our objective is different since our program follows the approximation of consistencies. Expected solutions are just some remarkable consistent states. It follows not surprisingly from this cross-fertilization that some CSP techniques are shared with the ones used in stable model systems such as SMODELS [18]. We are currently investigating this topic.

6 Implementation

We have implemented in Gnu-Prolog [8] a small translator called `csp2smodels` from CSPs to normal logic programs in order to feed the SMODELS [18] system. Here we present and illustrate experimentally our results.

A CSP is written in a simple syntax: each variable X must have a domain, which is expressed as an interval by `domain(x,`*`min`*`,`*`max`*`)` or given by a list of values `domain(x,[`v_1`, ..., `v_n`])`. A constraint may be described by a set of solution tuples by `constraint(`*`name`*`,[`*`list of variables`*`],[`*`list of tuples`*`])` or may be some of the SMODELS built-ins, for example $X = Y$ is written `constraint(eq,[x,y])`.

Example 32 (example 27 continued).
Our running example is written as follows:

```
domain(x,0,2).
domain(y,0,2).
domain(z,0,2).

constraint(k1,[x,y],[[0,1],[1,2],[2,1]]).
constraint(k2,[y,z],[[0,2],[1,1]]).
```

Here is an excerpt from the output program. First variable ranges are declared for SMODELS' grounding procedure:

```
domx(0..2).
#domain domx(X).
#hide domx(X).
```

User constraints are stated in extension:

```
k1(0,1).
k1(1,2).
k1(2,1).
#hide k1(X,Y).
```

Then the main program is written out, with for example rule-clause(k_1, X):

```
k1x(X) :- k1(X,Y), x(X), y(Y).
#hide k1x(X).
```

The variable Y is involved in two constraints appears negatively in two rules:

```
ny(Y) :- not k1y(Y).
ny(Y) :- not k2y(Y).
#hide ny(Y).
```

Then the last type of clause which ensure the stability of the models:

```
x(X) :- not nx(X).
```

Note that auxiliary names are hidden to focus on the useful information in the models, namely the predicates $x(X)$. The stable models, computed with `smodels 0` correspond to the following approximations computed by arc-consistency:

```
smodels version 2.27. Reading...done
Answer: 1
Stable Model:
Answer: 2
Stable Model: x(2) y(1) z(1)
Answer: 3
Stable Model: x(0) x(2) y(1) z(1)
Answer: 4
Stable Model: x(0) y(1) z(1)
False
```

The SMODELS system also finds the CSP's solutions as singletonic and consistent interpretations. The translator may be downloaded from the authors' web page.

7 Conclusion

In this paper, we presented a CLP logical scheme which allows to reflect the approximation made by arc-consistency in a declarative way. We first provide a definite CLP program and prove that the fixed-points of its associated operator are related to the computed approximation of consistencies. Then we transform it into an intuitive normal program and we show that the approximations computed by arc-consistency are its stable models. Moreover, the original CSP's solutions are given by a certain class of models we call singletonic.

By showing connections between CSP approximations and stable models, this work contributes to bridge the gap between traditional ways of solving CSPs by consistency and the knowledge representation paradigm of logic programming with stable model semantics.

References

1. K.R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1999.
2. K.R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.
3. K.R. Apt and R.N. Bol. Logic programming and negation : a survey. *Journal of Logic Programming*, 19/20:9–71, 1994.
4. Frédéric Benhamou. Heterogeneous constraint solving. In Michael Hanus and Mario Rodríguez-Artalejo, editors, *Algebraic and Logic Programming, 5th International Conference, ALP'96*, volume 1139 of *lncs*, pages 62–76, Aachen, Germany, sep 1996. Springer.

5. Thi-Bich-Hanh Dao, Arnaud Lallouet, Andrei Legtchenko, and Lionel Martin. Indexical-based solver learning. In Pascal van Hentenryck, editor, *International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *LNCS*, pages 541–555, Ithaca, NY, USA, Sept. 7 - 13 2002. Springer.
6. Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.
7. Rina Dechter and Judea Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.
8. Daniel Diaz and Philippe Codognet. The GNU Prolog system and its implementation. In *ACM Symposium on Applied Computing*, volume 2, pages 728–732, Villa Olmo, Como, Italy, 2000.
9. François Fages, Julian Fowler, and Thierry Sola. Experiments in reactive constraint logic programming. *Journal of Logic Programming*, 37(1-3):185–212, 1998.
10. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1-3):95–138, 1998.
11. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *International Joint Conference and Symposium on Logic Programming*, pages 1070–1080, Seattle, 1988. IEEE.
12. Joxan Jaffar, Michael Maher, Kim Marriott, and Peter Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1-3):1–46, 1998.
13. Arnaud Lallouet, Thi-Bich-Hanh Dao, and AbdelAli Ed-Dbali. Language, definition and optimal computation of CSP approximations. In Susan Haller and Ingrid Russell, editors, *Flairs'03, International Florida Artificial Intelligence Conference*, pages 182–186, St Augustine, FL, USA, 2003. AAAI Press.
14. Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
15. Victor W. Marek and Mirosław Truszczyński. *Stable Models and an Alternative Logic Programming Paradigm*, pages 375–378. Artificial Intelligence. Springer-Verlag, 1999.
16. Ugo Montanari and Francesca Rossi. Constraint relaxation may be perfect. *Artificial Intelligence*, 48:143–170, 1991.
17. Ilkka Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.
18. Ilkka Niemelä and Patrick Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *International Conference on Logic Programming and Non Monotonic Reasoning*, pages 420–429, Dagstuhl, Germany, July 1997. Springer-Verlag.
19. Nikolay Pelov and Maurice Bruynooghe. Extending constraint logic programming with open functions. In Maurizio Gabbrielli and Frank Pfenning, editors, *Principles and Practice of Declarative Programming*, pages 235–244, Montreal, Canada, 2000. ACM.
20. Nikolay Pelov, Emmanuel De Mot, and Marc Denecker. Logic programming approaches for representing and solving constraint satisfaction problems : A comparison. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning*, volume 1955 of *LNCS*, pages 225–239, Reunion Island, France, November 11-12 2000. Springer.
21. T. Przymusinski. Well-founded semantics coincides with three-valued stable semantics. *Fundamenta Informaticae*, XIII:445–463, 1990.
22. Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
23. P. van Hentenryck, V. Saraswat, and Y. Deville. Constraint processing in cc(fd). draft, 1991.
24. Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.