Automatic Generation of Constraint Propagation Algorithms for Small Finite Domains

Krzysztof R. Apt[1,2] and Eric Monfroy[1]

[1] CWI
P.O. Box 94079, 1090 GB Amsterdam, the Netherlands
{K.R.Apt,Eric.Monfroy}@cwi.nl
[2] University of Amsterdam, the Netherlands

Abstract. We study here constraint satisfaction problems that are based on predefined, explicitly given finite constraints. To solve them we propose a notion of *rule consistency* that can be expressed in terms of rules derived from the explicit representation of the initial constraints.
This notion of local consistency is weaker than arc consistency for constraints of arbitrary arity but coincides with it when all domains are unary or binary. For Boolean constraints rule consistency coincides with the closure under the well-known propagation rules for Boolean constraints.
By generalizing the format of the rules we obtain a characterization of arc consistency in terms of so-called inclusion rules. The advantage of rule consistency and this rule based characterization of the arc consistency is that the algorithms that enforce both notions can be automatically generated, as `CHR` rules. So these algorithms could be integrated into constraint logic programming systems such as ECL^{i}PS^e.
We illustrate the usefulness of this approach to constraint propagation by discussing the implementations of both algorithms and their use on various examples, including Boolean constraints, three valued logic of Kleene, constraints dealing with Waltz's language for describing polyhedreal scenes, and Allen's qualitative approach to temporal logic.

1 Introduction

In constraint programming the programming process is limited to a generation of constraints and a solution of the so obtained constraint satisfaction problems (CSP's) by general or domain dependent methods.

On the theoretical side several notions of local consistency, notably arc consistency for constraints of arbitrary arity, have been defined and various search methods have been proposed. On the practical side several constraint programming systems were designed and implemented that provide a substantial support for constraint programming. This support is usually provided in the form of specific built-in constructs that support search and constraint propagation. For example, the arc consistency is built in the ILOG Solver and is present in a library of the most recent version of ECL^{i}PS^e.

In this paper we study CSP's that are built out of predefined, explicitly given finite constraints. Such CSP's often arise in practice. Examples include Boolean constraints,

constraints dealing with Waltz's language for describing polyhedreal scenes, Allen's temporal logic, and constraints in any multi-valued logic.

In such situations it is natural to explore the structure of these explicitly given constraints first and to use this information to reduce the considered CSP to a simpler yet equivalent one. This information can be expressed in terms of rules. This leads to a local consistency notion called *rule consistency* that turns out to be weaker than arc consistency for constraints of arbitrary arity.

When the original domains are all unary or binary, rule consistency coincides with arc consistency. When additionally the predefined constraints are the truth tables of the Boolean connectives, these rules coincide with the well-known rules for Boolean constraints, sometimes called unit propagation rules (see, e.g. [3]). As a side effect, this shows that the unit propagation rules characterize arc consistency. Rule consistency is thus a generalization of the unit propagation to non-binary domains.

Next, we show that by generalizing the notion of rules to so-called inclusion rules, we obtain a notion of local consistency that coincides with arc consistency for constraints of arbitrary arity.

The advantage of the rule consistency and this rule based characterization of the arc consistency is that the algorithms that enforce them can be automatically generated and provided on the constraint programming language level. For example, the rules in question can be generated automatically and represented as rules of the `CHR` language of [6] that is part of the ECL^{i}PS^e system. (For a more recent and more complete overview of `CHR` see [5].)

Consequently, the implementations of the algorithms that achieve rule consistency and arc consistency for the considered CSP's are simply these automatically generated `CHR` programs. When combined with a labeling procedure such `CHR` programs constitute automatically derived decision procedures for these CSP's.

The availability of the algorithms that enforce rule consistency and arc consistency on the constraint programming language level further contributes to the automatization of the programming process within the constraint programming framework. In fact, in the case of such CSP's built out of predefined, explicitly given finite constraints the user does not need to write one's own `CHR` rules for the considered constraints and can simply adopt all or some of the rules that are automatically generated. In the final example of the paper we also show how using the rules and the inclusion rules, we can implement more powerful notions of local consistency.

Alternatively, the generated rules and inclusion rules could be fed into any of the generic *Chaotic Iteration* algorithms of [2] and made available in such systems as the ILOG solver. This would yield rule consistency and an alternative implementation of arc consistency.

The algorithms that for an explicitly given finite constraint generate the appropriate rules that characterize rule consistency and arc consistency have (unavoidably) a running time that is exponential in the number of constraint variables and consequently are in general impractical.

To test the usefulness of these algorithms for small finite domains we implemented them in ECL^{i}PS^e and successfully used them on several examples including the ones mentioned above. The fact that we could handle these examples shows that this approach is of practical value and can be used to automatically derive practical decision procedures for constraint satisfaction problems defined over small finite domains. Also it shows the usefulness of the `CHR` language for an automatic generation of constraint solvers and of decision procedures.

The rest of the paper is organized as follows. In the next section we formalize the concept that a CSP is built out of predefined constraints. Next, in Section 3 we introduce the notion of a rule, define the notion of rule consistency and discuss an algorithm that can be used to generate the minimal set of rules that characterize this notion of local consistency. Then, in Section 4 we compare rule consistency to arc consistency. In Section 5 we generalize the notion of rules to so-called inclusion rules and discuss an algorithm analogous to the one of Section 3. This entails a notion of local consistency that turns out to be equivalent to arc consistency. Finally, in Section 6 we discuss the implementation of both algorithms. They generate from an explicit representation of a finite constraint a set of `CHR` rules that characterize respectively rule consistency and arc consistency. We also illustrate the usefulness of these implementations by means of several examples. Due to lack of space all proofs are omitted.

2 CSP's Built out of Predefined Constraints

Consider a finite sequence of variables $X := x_1, \ldots, x_n$ where $n \geq 0$, with respective domains $\mathcal{D} := D_1, \ldots, D_n$ associated with them. So each variable x_i ranges over the domain D_i. By a *constraint* C on X we mean a subset of $D_1 \times \ldots \times D_n$. In this paper we consider only finite domains.

By a *constraint satisfaction problem*, in short CSP, we mean a finite sequence of variables X with respective domains $\mathcal{D}$, together with a finite set $\mathcal{C}$ of constraints, each on a subsequence of X. We write it as $\langle \mathcal{C} \; ; \; x_1 \in D_1, \ldots, x_n \in D_n \rangle$, where $X := x_1, \ldots, x_n$ and $\mathcal{D} := D_1, \ldots, D_n$.

Consider now an element $d := d_1, \ldots, d_n$ of $D_1 \times \ldots \times D_n$ and a subsequence $Y := x_{i_1}, \ldots, x_{i_\ell}$ of X. Then we denote by $d[Y]$ the sequence $d_{i_1}, \ldots, d_{i_\ell}$.

By a *solution* to $\langle \mathcal{C} \; ; \; x_1 \in D_1, \ldots, x_n \in D_n \rangle$ we mean an element $d \in D_1 \times \ldots \times D_n$ such that for each constraint $C \in \mathcal{C}$ on a sequence of variables X we have $d[X] \in C$. We call a CSP *consistent* if it has a solution.

Consider now a constraint C on a sequence of variables X. Given a subsequence Y of X by the *domain of* Y we mean the set of all tuples from $D_1 \times \ldots \times D_k$, where $D_1, \ldots, D_k$ are the respective domains of the variables from Y.

In the introduction we informally referred to the notion of a CSP "being built out of predefined, explicitly given finite constraints." Let us make now this concept formal. We need two auxiliary notions first, where in preparation for the next definition we already consider constraints together with the domains over which they are defined.

Definition 1.

- *Given a constraint* $C \subseteq D_1 \times \ldots \times D_n$ *and a permutation* π *of* $[1..n]$ *we denote by* C^π *the relation defined by*

$$(a_1, \ldots, a_n) \in C^\pi \textit{ iff } (a_{\pi(1)}, \ldots, a_{\pi(n)}) \in C.$$

– *Given two constraints* $C \subseteq D_1 \times \ldots \times D_n$ *and* $E \subseteq D'_1 \times \ldots \times D'_n$ *we say that* C is based on E *if*
  - $D_i \subseteq D'_i$ *for* $i \in [1..n]$,
  - $C = E \cap (D_1 \times \ldots \times D_n)$. □

So the notion of "being based on" involves the domains of both constraints. If C is based on E, then C is the restriction of E to the domains over which C is defined.

Definition 2. *We assume that the "predefined constraints" are presented as a given in advance CSP* $\mathcal{BASE}$ *and the considered CSP* $\mathcal{P}$ *is related to* $\mathcal{BASE}$ *as follows:*

– *There is a mapping* f *that relates each constraint* C *of* $\mathcal{P}$ *to a constraint* $f(C)$ *of* $\mathcal{BASE}$.
– *Each constraint* C *of* $\mathcal{P}$ *is based on* $f(C)^\pi$, *where* π *is a permutation of* $[1..n]$ *and* n *the arity of* C.

We say then that $\mathcal{P}$ *is* based on $\mathcal{BASE}$. □

In the above definition the "permuted" relations R^π allow us to abstract from the variable ordering used in $\mathcal{BASE}$. The following example illustrates this notion.

Example 1. Consider the well-known full adder circuit. It is defined by the following formula:

$$add(i_1, i_2, i_3, o_1, o_2) \equiv$$
$$xor(i_1, i_2, x_1), and(i_1, i_2, a_1), xor(x_1, i_3, o_2), and(i_3, x_1, a_2), or(a_1, a_2, o_1),$$

where and, xor and or are defined in the expected way. We can view the original constraints as the following CSP:

$$\mathcal{BOOL} := \langle and(x, y, z), xor(x, y, z), or(x, y, z)\ ;\ x \in \{0, 1\}, y \in \{0, 1\}, z \in \{0, 1\}\rangle.$$

$\mathcal{BOOL}$ should be viewed just as an "inventory" of the predefined constraints and not as a CSP to be solved. Now, any query concerning the full adder can be viewed as a CSP based on $\mathcal{BOOL}$. For example, in Section 6 we shall consider the query $add(1, x, y, z, 0)$. It corresponds to the following CSP based on $\mathcal{BOOL}$:

$$\langle\ xor(i_1, i_2, x_1),\ and(i_1, i_2, a_1),\ xor(x_1, i_3, o_2),\ and(i_3, x_1, a_2),\ or(a_1, a_2, o_1)\ ;$$
$$i_1 \in \{1\}, i_2 \in \{0, 1\}, i_3 \in \{0, 1\}, o_1 \in \{0, 1\}, o_2 \in \{0\}, a_1 \in \{0, 1\}, a_2 \in \{0, 1\},$$
$$x_1 \in \{0, 1\}\ \rangle.$$

□

3 Rule Consistency

Our considerations crucially rely on the following notion of a rule.

Definition 3. *Consider a constraint* C *on a sequence of variables VAR, a subsequence* X *of VAR and a variable* y *of VAR not in* X, *a tuple* s *of elements from the domain of* X *and an element* a *from the domain of* y. *We call* $X = s \rightarrow y \neq a$ *a* rule *(for* C*).*

– *We say that $X = s \rightarrow y \neq a$ is* valid *(*for *C) if for every tuple $d \in C$ the equality $d[X] = s$ implies $d[y] \neq a$.*
– *We say that $X = s \rightarrow y \neq a$ is* feasible *(*for *C) if for some tuple $d \in C$ the equality $d[X] = s$ holds.*
– *Suppose that $X := x_1, \ldots, x_k$ and $s := s_1, \ldots, s_k$. We say that C is* closed under the rule *$X = s \rightarrow y \neq a$ if the fact that the domain of each variable x_j equals $\{s_j\}$ implies that a is not an element of the domain of the variable y.*

Further, given a sequence of variables Z that extends X and a tuple of elements u from the domain of Z that extends s, we say that the rule $Z = u \rightarrow y \neq a$ extends *$X = s \rightarrow y \neq a$. We call a rule* minimal *if it is feasible and it does not properly extend a valid rule.* □

Note that rules that are not feasible are trivially valid. To illustrate the introduced notions consider the following example.

Example 2. Take as a constraint the ternary relation that represents the conjunction $and(x, y, z)$. It can be viewed as the following relation:

$$\{(0, 0, 0), (0, 1, 0), (1, 0, 0), (1, 1, 1)\}.$$

In other words, we assume that each of the variables x, y, z has the domain $\{0, 1\}$ and view $and(x, y, z)$ as the constraint on x, y, z that consists of the above four triples.

It is easy to see that the rule $x = 0 \rightarrow z \neq 1$ is valid for $and(x, y, z)$. Further, the rule $x = 0, y = 1 \rightarrow z \neq 1$ extends the rule $x = 0 \rightarrow z \neq 1$ and is also valid for $and(x, y, z)$. However, out of these two rules only $x = 0 \rightarrow z \neq 1$ is minimal.

Finally, both rules are feasible while the rules $x = 0, z = 1 \rightarrow y \neq 0$ and $x = 0, z = 1 \rightarrow y \neq 1$ are not feasible. □

Note that a rule that extends a valid rule is valid, as well. So validity extends "upwards".

Next, we introduce a notion of local consistency that is expressed in terms of rules.

Definition 4. *Consider a CSP $\mathcal{P}$ is based on a CSP $\mathcal{BASE}$. Let C be a constraint of $\mathcal{P}$ on the variables $x_1, \ldots, x_n$ with respective non-empty domains $D_1, \ldots, D_n$. For some constraint $f(C)$ of $\mathcal{BASE}$ and a permutation π we have $C = f(C)^\pi \cap (D_1 \times \ldots \times D_n)$.*

– *We call the constraint C* rule consistent *(w.r.t. $\mathcal{BASE}$) if it is closed under all rules that are valid for $f(C)^\pi$.*
– *We call a CSP* rule consistent *(w.r.t. $\mathcal{BASE}$) if all its constraints are rule consistent.*

□

In what follows we drop the reference to $\mathcal{BASE}$ if it is clear from the context.

Example 3. Take as the base CSP

$$\mathcal{BASE} := \langle and(x, y, z) \; ; \; x \in \{0, 1\}, y \in \{0, 1\}, z \in \{0, 1\}\rangle$$

and consider the following four CSP's based on it:

1. $\langle and(x, y, z) \ ; \ x \in \{0\}, y \in D_y, z \in \{0\}\rangle$,
2. $\langle and(x, y, z) \ ; \ x \in \{1\}, y \in D_y, z \in \{0, 1\}\rangle$,
3. $\langle and(x, y, z) \ ; \ x \in \{0, 1\}, y \in D_y, z \in \{1\}\rangle$,
4. $\langle and(x, y, z) \ ; \ x \in \{0\}, y \in D_y, z \in \{0, 1\}\rangle$,

where D_y is a subset of $\{0, 1\}$. We noted in Example 2 that the rule $x = 0 \rightarrow z \neq 1$ is valid for $and(x, y, z)$. In the first three CSP's its only constraint is closed under this rule, while in the fourth one not since 1 is present in the domain of z whereas the domain of x equals $\{0\}$. So the fourth CSP is not rule consistent. One can show that the first two CSP's are rule consistent, while the third one is not since it is not closed under the valid rule $z = 1 \rightarrow x \neq 0$.

The following observation is useful.

Note 1. Consider two constraints C and E such that $C \subseteq E$. Then C is closed under all valid rules for E iff it is closed under all minimal valid rules for E.

This allows us to confine our attention to minimal valid rules. We now introduce an algorithm that given a constraint generates the set of all minimal valid rules for it. We collect the generated rules in a list. We denote below the empty list by **empty** and the result of insertion of an element r into a list L by $\mathbf{insert}(r, L)$.

By an *assignment* to a sequence of variables X we mean here an element s from the domain of X such that for some $d \in C$ we have $d[X] = s$. Intuitively, if we represent the constraint C as a table with rows corresponding to the elements (tuples) of C and the columns corresponding to the variables of C, then an assignment to X is a tuple of elements that appears in some row in the columns that correspond to the variables of X. This algorithm has the following form where we assume that the considered constraint C is defined on a sequence of variables *VAR* of cardinality n.

RULES GENERATION algorithm

```
L := empty;
FOR i:= 0 TO n-1 DO
   FOR each subset X of VAR of cardinality i DO
      FOR each assignment s to X DO
         FOR each y in VAR-X DO
            FOR each element d from the domain of y DO
               r := X = s → y ≠ d;
               IF r is valid for C
                  and it does not extend an element of L
                  THEN insert(r, L)
```

The following result establishes correctness of this algorithm.

Theorem 1. *Given a constraint C the* RULES GENERATION *algorithm produces in L the set of all minimal valid rules for C.*

Note that because of the minimality property no rule in L extends another.

4 Relating Rule Consistency to Arc Consistency

To clarify the status of rule consistency we compare it now to the notion of arc consistency. This notion was introduced in [8] for binary relations and was extended to arbitrary relations in [9]. Let us recall the definition.

Definition 5.

- *We call a constraint C on a sequence of variables X* arc consistent *if for every variable x in X and an element a in its domain there exists $d \in C$ such that $a = d[x]$. That is, each element in each domain participates in a solution to C.*
- *We call a CSP* arc consistent *if all its constraints are arc consistent.* □

The following result relates for constraints of arbitrary arity arc consistency to rule consistency.

Theorem 2. *Consider a CSP $\mathcal{P}$ based on a CSP $\mathcal{BASE}$. If $\mathcal{P}$ is arc consistent then it is rule consistent w.r.t. $\mathcal{BASE}$.*

The converse implication does not hold in general as the following example shows.

Example 4. Take as the base the following CSP

$$\mathcal{BASE} := \langle C \; ; \; x \in \{0,1,2\}, y \in \{0,1,2\}\rangle$$

where the constraint C on x, y that equals the set $\{(0,1),(1,0),(2,2)\}$. So C can be viewed as the following relation:

$$\{(0,1),(1,0),(2,2)\}.$$

Next, take for D_1 the set $\{0,1\}$ and D_2 the set $\{0,1,2\}$. Then the CSP $\langle C \cap (D_1 \times D_2) \; ; \; x \in D_1, y \in D_2\rangle$, so $\langle\{(0,1),(1,0)\} \; ; \; x \in \{0,1\}, y \in \{0,1,2\}\rangle$ is based on $\mathcal{BASE}$ but is not arc consistent since the value 2 in the domain of y does not participate in any solution. Yet, it is easy to show that the only constraint of this CSP is closed under all rules that are valid for C. □

However, if each domain has at most two elements, then the notions of arc consistency and rule consistency coincide. More precisely, the following result holds.

Theorem 3. *Let $\mathcal{BASE}$ be a CSP each domain of which is unary or binary. Consider a CSP $\mathcal{P}$ based on $\mathcal{BASE}$. Then $\mathcal{P}$ is arc consistent iff it is rule consistent w.r.t. $\mathcal{BASE}$.*

5 Inclusion Rule Consistency

We saw in the previous section that the notion of rule consistency is weaker than that of arc consistency for constraints of arbitrary arity. We now show how by modifying the format of the rules we can achieve arc consistency. To this end we introduce the following notions.

Definition 6. *Consider a constraint C over a sequence variables VAR, a subsequence $X := x_1, \ldots, x_k$ of VAR and a variable y of VAR not in X, a tuple $S := S_1, \ldots, S_k$ of respective subsets of the domains of the variables from X and an element a from the domain of y.*

We call $X \subseteq S \rightarrow y \neq a$ an inclusion rule *(*for *C). We say that $X \subseteq S \rightarrow y \neq a$ is* valid *(*for *C) if for every tuple $d \in C$ the fact that $d[x_i] \in S_i$ for $i \in [1..k]$ implies that $d[y] \neq a$ and that $X \subseteq S \rightarrow y \neq a$ is* feasible *(*for *C) if for some tuple $d \in C$ we have $d[x_i] \in S_i$ for $i \in [1..k]$.*

Further, we say that a constraint C is closed under the inclusion rule *$X \subseteq S \rightarrow y \neq a$ if the fact that the domain of each variable x_j is included in S_j implies that a is not an element of the domain of the variable y.* □

By choosing in the above definition singleton sets $S_1, \ldots, S_k$ we see that the inclusion rules generalize the rules of Section 3. Note that inclusion rules that are not feasible are trivially valid.

In analogy to Definition 4 we now introduce the following notion.

Definition 7. *Consider a CSP $\mathcal{P}$ is based on a CSP $\mathcal{BASE}$. Let C be a constraint of $\mathcal{P}$ on the variables $x_1, \ldots, x_n$ with respective non-empty domains $D_1, \ldots, D_n$. For some constraint $f(C)$ of $\mathcal{BASE}$ and a permutation π we have $C = f(C)^\pi \cap (D_1 \times \ldots \times D_n)$.*

- *We call the constraint C* inclusion rule consistent *(w.r.t. $\mathcal{BASE}$) if it is closed under all inclusion rules that are valid for $f(C)^\pi$.*
- *We call a CSP* inclusion rule consistent *(w.r.t. $\mathcal{BASE}$) if all its constraints are inclusion rule consistent.* □

We now have the following result.

Theorem 4. *Consider a CSP $\mathcal{P}$ based on a CSP $\mathcal{BASE}$. Then $\mathcal{P}$ is arc consistent iff it is inclusion rule consistent w.r.t. $\mathcal{BASE}$.*

Example 4 shows that the notions of rule consistency and inclusion rule consistency do not coincide.

In Section 3 we introduced an algorithm that given a constraint C generated the set of all minimal rules valid for C. We now modify it to deal with the inclusion rules. First we need to adjust the notions of an extension and of minimality.

Definition 8. *Consider a constraint C on a sequence of variables VAR. Let $X := x_1, \ldots, x_k$ and $Z := z_1, \ldots, z_\ell$ be two subsequences of VAR such that Z extends X and y a variable of VAR not in Z. Further, let $S := S_1, \ldots, S_k$ be the sequence of respective subsets of the domains of the variables from X, $U := U_1, \ldots, U_\ell$ the sequence of respective subsets of the domains of the variables from Z, and a an element from the domain of y.*

We say that the inclusion rule $r_1 := Z \subseteq U \rightarrow y \neq a$ extends *$r_2 := X \subseteq S \rightarrow y \neq a$ if for each common variable of X and Z the corresponding element of U is a subset of the corresponding element of S. We call an inclusion rule* minimal *if it is feasible and it does not properly extend a valid inclusion rule.* □

To clarify these notions consider the following example.

Example 5. Consider a constraint on variables x, y, z, each with the domain $\{+, -, l, r\}$, that is defined by the following relation:

$$\{(+,+,+), (-,-,-), (l,r,-), (-,l,r), (r,-,l)\}$$

This constraint is the so-called *fork* junction in the language of [10] for describing polyhedreal scenes. Note that the following three inclusion rules

$$r_1 := x \subseteq \{+,-\} \rightarrow z \neq l,$$

$$r_2 := x \subseteq \{+\} \rightarrow z \neq l,$$

and

$$r_3 := x \subseteq \{-\}, y \subseteq \{l\} \rightarrow z \neq l$$

are all valid. Then the inclusion rules r_2 and r_3 extend r_1 while the inclusion rule r_1 extends neither r_2 nor r_1. Further, the inclusion rules r_2 and r_3 are incomparable in the sense that none extends the other. □

The following counterpart of Note 1 holds.

Note 2. Consider two constraints C and E such that $C \subseteq E$. Then C is closed under all valid inclusion rules for E iff it is closed under all minimal valid inclusion rules for E.

As in Section 3 we now provide an algorithm that given a constraint generates the set of all minimal valid inclusion rules. We assume here that the considered constraint C is defined on a sequence of variables *VAR* of cardinality n.

Instead of assignments that are used in the RULES GENERATION algorithm we now need a slightly different notion. To define it for each variable x from *VAR* we denote the set $\{d[x] \mid d \in C\}$ by $C[x]$. By a *weak assignment* to a sequence of variables $X := x_1, \ldots, x_k$ we mean here a sequence $S_1, \ldots, S_k$ of subsets of, respectively, $C[x_1], \ldots, C[x_k]$ such that some $d \in C$ exists such that $d[x_i] \in S_i$ for each $i \in [1..k]$.

Intuitively, if we represent the constraint C as a table with rows corresponding to the elements of C and the columns corresponding to the variables of C and we view each column as a set of elements, then a weak assignment to X is a tuple of subsets of the columns that correspond to the variables of X that "shares" an assignment.

In the algorithm below the weak assignments to a fixed sequence of variables are considered in decreasing order in the sense that if the weak assignments $S_1, \ldots, S_k$ and $U_1, \ldots, U_k$ are such that for $i \in [1..k]$ we have $U_i \subseteq S_i$, then $S_1, \ldots, S_k$ is considered first.

INCLUSION RULES GENERATION algorithm

```
L := empty;
FOR i:= 0 TO n-1 DO
  FOR each subset X of VAR of cardinality i DO
    FOR each weak assignment S to X in decreasing order DO
      FOR each y in VAR-X DO
        FOR each element d from the domain of y DO
```

```
r :=  X ⊆ S  →  y ≠ d;
IF r is valid for C
   and it does not extend an element of L
   THEN  insert(r, L)
```

The following result establishes correctness of this algorithm.

Theorem 5. *Given a constraint C the* INCLUSION RULES GENERATION *algorithm produces in L the set of all minimal valid inclusion rules for C.*

6 Applications

In this section we discuss the implemention of the RULES GENERATION and INCLUSION RULES GENERATION algorithms and discuss their use on selected domains.

6.1 Constraint Handling Rules (CHR)

In order to validate our approach we have realized in the Prolog platform ECL^{i}PS^e a prototype implementation of both the RULES GENERATION algorithm and the INCLUSION RULES GENERATION algorithm. These implementations generate CHR rules that deal with finite domain variables using an ECL^{i}PS^e library.

Constraint Handling Rules (CHR) of [6] is a declarative language that allows one to write guarded rules for rewriting constraints. These rules are repeatedly applied until a fixpoint is reached. The rule applications have a precedence over the usual resolution step of logic programming.

CHR provides two types of rules: simplification rules that replace a constraint by a simpler one, and propagation rules that add new constraints.

Our rules and inclusion rules can be modelled by means of propagation rules. To illustrate this point consider some constraint $cons$ on three variables, A, B, C, each with the domain $\{0, 1, 2\}$.

The RULES GENERATION algorithm generates rules such as $(A, C) = (0, 1) \rightarrow B \neq 2$. This rule is translated into a CHR rule of the form: `cons(0,B,1) ==> B##2`. Now, when a constraint in the program query matches `cons(0,B,1)`, this rule is fired and the value 2 is removed from the domain of the variable `B`.

In turn, the INCLUSION RULES GENERATION algorithm generates rules such as $(A, C) \subseteq (\{0\}, \{1, 2\}) \rightarrow B \neq 2$. This rule is translated into the CHR rule

```
cons(0,B,C) ==>in(C,[1,2]) | B##2
```

where the `in` predicate is defined by

```
in(X,L):- dom(X,D), subset(D,L).
```

So `in(X,L)` holds if the current domain of the variable `X` (yielded by the built-in `dom` of ECL^{i}PS^e) is included in the list `L`.

Now, when a constraint matches `cons(0,B,C)` *and* the current domain of the variable `C` is included in `[1,2]`, the value 2 is removed from the domain of `B`. So for both types of rules we achieve the desired effect.

In the examples below we combine the rules with the same premise into one rule in an obvious way and present these rules in the CHR syntax.

6.2 Generating the rules

We begin by discussing the generation of rules and inclusion rules for some selected domains. The times given refer to an implementation ran on a Silicon Graphics O2 with 64 Mbytes of memory and a 180 MHZ processor.

Boolean constraints As the first example consider the Boolean constraints, for example the conjunction constraint `and(X,Y,Z)` of Example 2. The RULES GENERATION algorithm generated in 0.02 seconds the following six rules:

```
and(1,1,X) ==> X##0.
and(X,0,Y) ==> Y##1.
and(0,X,Y) ==> Y##1.
and(X,Y,1) ==> X##0,Y##0.
and(1,X,0) ==> X##1.
and(X,1,0) ==> X##1.
```

Because the domains are here binary we can replace the conclusions of the form `U ## 0` by `U = 1` and `U ## 1` by `U = 0`. These become then the well-known rules that can be found in [5, page 113].

In this case, by virtue of Theorem 3, the notions of rule and arc consistency coincide, so the above six rules characterize the arc consistency of the `and` constraint. Our implementations of the RULES GENERATION and the INCLUSION RULES GENERATION algorithms yield here the same rules.

Three valued logics Next, consider the three valued logic of [7, page 334] that consists of three values, t (true), f (false) and u (unknown). We only consider here the crucial equivalence relation $\equiv$ defined by the truth table

$\equiv$	t	f	u
t	t	f	u
f	f	t	u
u	u	u	u

that determines a ternary constraint with nine triples. We obtain for it 20 rules and 26 inclusion rules. Typical examples are

`equiv(X,Y,f) ==> X##u,Y##u.`
and
`equiv(t,X,Y) ==> in(Y,[f, u])` | `X##t.`

Waltz' language for describing polyhedreal scenes Waltz' language consists of four constraints. One of them, the fork junction was already mentioned in Example 5. The RULES GENERATION algorithm generated for it 12 rules and the INCLUSION RULES GENERATION algorithm 24 inclusion rules.

Another constraint, the so-called T junction, is defined by the following relation:

$$\{(r,l,+),(r,l,-),(r,l,r),(r,l,l)\}.$$

In this case the RULES GENERATION algorithm and the INCLUSION RULES GENERATION algorithm both generate the same output that consists of just one rule:

```
t(X,Y,Z) ==> X##'l',X##'-',X##'+',Y##'r',Y##'-',Y##'+'.
```

So this rule characterizes both rule consistency and arc consistency for the CSP's based on the T junction.

For the other two constraints, the L junction and the arrow junction, the generation of the rules and inclusion rules is equally straightforward.

6.3 Using the rules

Next, we show by means of some examples how the generated rules can be used to reduce or to solve specific queries. Also, we show how using compound constraints we can achieve local consistency notions that are stronger than arc consistency for constraints of arbitrary arity.

Waltz' language for describing polyhedreal scenes The following predicate describes the impossible object given in Figure 12.18 of [11, page 262]:

```
imp(AF,AI,AB,IJ,IH,JH,GH,GC,GE,EF,ED,CD,CB):-
     S1=[AF,AI,AB,IJ,IH,JH,GH,GC,GE,EF,ED,CD,CB],
     S2=[FA,IA,BA,JI,HI,HJ,HG,CG,EG,FE,DE,DC,BC],
     append(S1,S2,S), S :: [+,-,l,r],

     arrow(AF,AB,AI), l(BC,BA), arrow(CB,CD,CG),
     l(DE,DC), arrow(ED,EG,EF), l(FA,FE), fork(GH,GC,GE),
     arrow(HG,HI,HJ), fork(IA,IJ,IH), l(JH,JI),

     line(AF,FA), line(AB,BA), line(AI,IA), line(IJ,JI),
     line(IH,HI), line(JH,HJ), line(GH,HG), line(FE,EF),
     line(GE,EG), line(GC,CG), line(DC,CD), line(ED,DE),
     line(BC,CB).
```

where the supplementary constraint `line` is defined by the following relation:

$$\{(+,+),(-,-),(l,r),(r,l)\}$$

When using the rules obtained by the RULES GENERATION algorithm and associated with the `fork`, `arrow`, `t`, `l`, and `line` constraints, the query

```
imp(AF,AI,AB,IJ,IH,JH,GH,GC,GE,EF,ED,CD,CB)
```

reduces in 0.009 seconds the variable domains to `AF ∈ [+,-, l]`, `AI ∈ [+,-]`, `AB ∈ [+,-,r]`, `IJ ∈ [+,-,l,r]`, `IH ∈ [+,-,l,r]`, `JH ∈ [+,-,l,r]`, `GH ∈ [+,-,l,r]`, `GC ∈ [+,-,l,r]`, `GE ∈ [+,-,l,r]`, `EF ∈ [+,-]`, `ED ∈ [+,-,l]`, `CD ∈ [+,-,r]`, and `CB ∈ [+,-,l]`.

But some constraints remain unsolved, so we need to add a labeling mechanism to prove the inconsistency of the problem. On the other hand, when using the inclusion rules, the inconsistency is detected without any labeling in 0.06 seconds.

In the well-known example of the cube given in Figure 12.15 of [11, page 260] the inclusion rules are also more powerful than the rules and both sets of rules reduce the problem but in both cases labeling is needed to produce all four solutions.

Temporal reasoning In [1] approach to temporal reasoning the entities are intervals and the relations are temporal binary relations between them. [1] found that there are 13 possible temporal relations between a pair of events, namely `before, during, overlaps, meets, starts, finishes`, the symmetric relations of these six relations and `equal`. We denote these 13 relations respectively by `b`, `d`, `o`, `m`, `s`, `f`, `b-`, `d-`, `o-`, `m-`, `s-`, `f-`, `e` and their set by `TEMP`.

Consider now three events, `A, B` and `C` and suppose that we know the temporal relations between the pairs `A` and `B`, and `B` and `C`. The question is what is the temporal relation between `A` and `C`. To answer it [1] provided a 13×13 table. This table determines a ternary constraint between a triple of events, `A, B` and `C` that we denote by `tr`. For example,

$$(\texttt{overlaps}, \texttt{before}, \texttt{before}) \in \texttt{tr}$$

since `A overlaps B` and `B` is `before C` implies that `A` is `before C`.

Using this table, the RULE GENERATION algorithm produced for the constraint `tr` 498 rules in 31.16 seconds.

We tried this set of rules to solve the following problem from [1]: "John was not in the room when I touched the switch to turn on the light.". We have here three events: `S`, the time of touching the switch; `L`, the time the light was on; and `J`, the time that John was in the room. Further, we have two relations: `R1` between `L` and `S`, and `R2` between `S` and `J`. This problem is translated into the CSP $\langle \texttt{tr} \; ; \; \texttt{R1} \in [\texttt{o}-, \texttt{m}-], \texttt{R2} \in [\texttt{b}, \texttt{m}, \texttt{b}-, \texttt{m}-], \texttt{R3} \in \texttt{TEMP} \rangle$, where `tr` is the above constraint on the variables `R1`, `R2`, `R3`.

To infer the relation `R3` between `L` and `J` we can use the following query [1]:

```
R1::[o-,m-], R2::[b,m,b-,m-],
R3::[b,d,o,m,s,f,b-,d-,o-,m-,s-,f-,e],
tr(R1,R2,R3), labeling([R1,R2,R3]).
```

We then obtain the following solutions in 0.06 seconds: `(R1,R2,R3)` $\in$ { `(m-,b,b), (m-,b,d-), (m-,b,f-), (m-,b,m), (m-,b,o), (m-,b-,b-), (m-,m,e), (m-,m,s), (m-,m,s-), (m-,m-,b-), (o-,b,b), (o-,b,d-), (o-,b,f-), (o-,b,m), (o-,b,o), (o-,b-,b-), (o-,m,d-), (o-,m,f-), (o-,m,o), (o-,m-,b-)` }.

To carry on (as in [1]), we now complete the problem with: "But John was in the room later while the light went out.". This is translated into: "`L overlaps`, `starts`, or `is during J`", i.e., `R3` $\in$ `[o,s,d]`.

We now run the following query:

```
R1::[o-,m-], R2::[b,m,b-,m-], R3::[o,s,d],
tr(R1,R2,R3), labeling([R1,R2,R3]).
```

and obtain four solutions in 0.04 seconds:
`(R1,R2,R3)` $\in$ { `(m-,b,o), (m-,m,s), (o-,b,o), (o-,m,o)` }.

[1] Since no variable is instantiated, we need to perform labeling to effectively apply the rules.

Full adder This final example illustrates how we can use the rules and the inclusion rules to implement more powerful notions of local consistency. The already discussed in Example 1 full adder circuit can be defined by the following constraint logic program (see, e.g., [5]) that uses the Boolean constraints `and`, `xor` and `or`:

```
add(I1,I2,I3,O1,O2):-
        [I1,I2,I3,O1,O2,A1,A2,X1]:: 0..1,
        xor(I1,I2,X1), and(I1,I2,A1), xor(X1,I3,O2),
        and(I3,X1,A2), or(A1,A2,O1).
```

The query `add(I1,I2,I3,O1,O2)` followed by a labeling mechanism generates the explicit definition (truth table) of the `full_adder` constraint with eight entries such as `full_adder(1,0,1,1,0).`

We can now generate rules and inclusion rules for the compound constraint (here the `full_adder` constraint) that is defined by means of some basic constraints (here the `and, or` and `xor` constraints). These rules refer to the compound constraint and allow us to reason about it directly instead of by using the rules that deal with the basic constraints.

In the case of the `full_adder` constraint the RULES GENERATION algorithm generated 52 rules in 0.27 seconds. The constraint propagation carried out by means of these rules is more powerful than the one carried out by means of the rules generated for the `and, or` and `xor` constraints.
For example, the query `[X,Y,Z]::[0,1], full_adder(1,X,Y,Z,0)` reduces `Z` to 1 whereas the query `[X,Y,Z]::[0,1], add(1,X,Y,Z,0)` does not reduce `Z` at all.

This shows that the rule consistency for a compound constraint defined by means of the basic constraints is in general stronger than the rule consistency for the basic constraints treated separately. In fact, in the above case the rules for the `full_adder` constraint yield the relational (1,5)-consistency notion of [4], whereas by virtue of Theorem 3, the rules for the `and, or` and `xor` constraints yield a weaker notion of arc consistency.

7 Conclusions

The aim of this paper was to show that constraint satisfaction problems built out of explicitly given constraints defined over small finite domains can be often solved by means of automatically generated constraint propagation algorithms.

We argued that such CSP's often arise in practice and consequently the methods here developed can be of practical use. Currently we are investigating how the approach of this paper can be applied to a study of various decision problems concerning specific multi-valued logics and how this in turn could be used for an analysis of digital circuits. Other applications we are now studying involve non-linear constraints over small finite domains and the analysis of polyhedreal scenes in presence of shadows (see [10]).

The introduced notion of rule consistency is weaker than arc consistency and can be in some circumstances the more appropriate one to use. For example, for the case of temporal reasoning considered in the last section we easily generated all 498 rules that enforce rule consistency whereas 24 hours turned out not be enough to generate the inclusion rules that enforce arc consistency.

Finally, the notions of rule consistency and inclusion rule consistency could be parametrized by the desired maximal number of variables used in the rule premises. Such parametrized versions of these notions could be useful when dealing with constraints involving a large number of variables. Both the RULES GENERATION algorithm and the INCLUSION RULES GENERATION algorithm and their implementations can be trivially adapted to such parametrized notions.

The approach proposed in this paper could be easily integrated into constraint logic programming systems such as ECL^{i}PS^e. This could be done by providing an automatic constraint propagation by means of the rules or the inclusion rules for flagged predicates that are defined by a list of ground facts, much in the same way as now constraint propagation for linear constraints over finite systems is automatically provided.

Acknowledgements

We would like to thank Thom Frühwirth, Andrea Schaerf and the anonymous referees for useful suggestions concerning this paper.

References

1. J.F. Allen. Maintaining knowledge about temporal intervals. *Communications of ACM*, 26(11):832–843, 1983.
2. K. R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1–2):179–210, 1999. Available via `http://xxx.lanl.gov/archive/cs/`.
3. M. Dalal. Efficient Propositional Constraint Propagation. In *Proceedings of the* 10^{th} *National Conference on Artificial Intelligence, AAAI'92*, pages 409–414, 1992. San Jose, California.
4. R. Dechter and P. van Beek. Local and global relational consistency. *Theoretical Computer Science*, 173(1):283–308, 20 February 1997.
5. T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998. Special Issue on Constraint Logic Programming (P. Stuckey and K. Marriot, Eds.).
6. Thom Frühwirth. Constraint Handling Rules. In Andreas Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 910, pages 90–107. Springer-Verlag, 1995. (Châtillon-sur-Seine Spring School, France, May 1994).
7. S. C. Kleene. *Introduction to Metamathematics*. van Nostrand, New York, 1952.
8. A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
9. R. Mohr and G. Masini. Good old discrete relaxation. In Y. Kodratoff, editor, *Proceedings of the 8th European Conference on Artificial Intelligence (ECAI)*, pages 651–656. Pitman Publishers, 1988.
10. D. L. Waltz. Generating semantic descriptions from drawings of scenes with shadows. In P. H. Winston, editor, *The Psychology of Computer Vision*. McGraw Hill, 1975.
11. P.H. Winston. *Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, third edition, 1992.