TLP 1 (6): 713–750, November 2001. © 2001 Cambridge University Press
DOI: 10.1017/S1471068401000072 Printed in the United Kingdom

*Constraint programming viewed as rule-based programming**

KRZYSZTOF R. APT
CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
and University of Amsterdam, The Netherlands
(*e-mail:* K.R.Apt@cwi.nl)

ERIC MONFROY
Institut de Recherche en Informatique de Nantes (IRIN), Université de Nantes,
2, rue de la Houssinière, BP 92208, 44322 Nantes Cedex 03, France
(*e-mail:* Eric.Monfroy@irin.univ-nantes.fr)

Abstract

We study here a natural situation when constraint programming can be entirely reduced to rule-based programming. To this end we explain first how one can compute on constraint satisfaction problems using rules represented by simple first-order formulas. Then we consider constraint satisfaction problems that are based on predefined, explicitly given constraints. To solve them we first derive rules from these explicitly given constraints and limit the computation process to a repeated application of these rules, combined with labeling. We consider two types of rule here. The first type, that we call equality rules, leads to a new notion of local consistency, called *rule consistency* that turns out to be weaker than arc consistency for constraints of arbitrary arity (called hyper-arc consistency in Marriott & Stuckey (1998)). For Boolean constraints rule consistency coincides with the closure under the well-known propagation rules for Boolean constraints. The second type of rules, that we call membership rules, yields a rule-based characterization of arc consistency. To show feasibility of this rule-based approach to constraint programming, we show how both types of rules can be automatically generated, as CHR rules of Frühwirth (1995). This yields an implementation of this approach to programming by means of constraint logic programming. We illustrate the usefulness of this approach to constraint programming by discussing various examples, including Boolean constraints, two typical examples of many valued logics, constraints dealing with Waltz's language for describing polyhedral scenes, and Allen's qualitative approach to temporal logic.

KEYWORDS: constraint programming, rule-based programming, finite domain

* A preliminary version of this article appeared in Apt & Monfroy (1999). In this version, we also present a framework for computing with rules on constraint satisfaction problems and discuss in detail the results of various experiments.

1 Introduction

1.1 Background

This paper is concerned with two styles of programming: constraint programming and rule-based programming. In constraint programming, the programming process is limited to a generation of constraints and a solution of the so obtained Constraint Satisfaction Problems (CSPs) by general or domain dependent methods. In rule-based programming, the programming process consists of a repeated application of rules. A theoretical basis for this programming paradigm consists of so-called production rules that were introduced in the 1970s, e.g. see Luger & Stubblefield (1998, pp. 171–186), though the idea goes back to the works of A. Thue and of E. Post in the first half of the twentieth century. The production rules are condition-action pairs, where the condition part is used to determine whether the rule is applicable, and the action part defines the action to be taken. The best known programming language built around this programming paradigm was OPS5 (Forgy, 1981).

Recently, there has been a revival of interest in rule-based programming in the context of constraint programming. The earliest example is the CHR language of Frühwirth (1995) that is a part of the ECL^{i}PS^e system. (For a more recent and more complete overview of CHR see Frühwirth, 1998.) The CHR rules extend the syntax of constraint logic programming by allowing two atoms in the conclusion and employing guards. These rules are predominantly used to write constraint solvers.

Another example of a programming language in which rules play an important role is ELAN. It offers a logical environment for specifying and prototyping deduction systems by means of conditional rewrite rules controlled by strategies. ELAN is used to support the design of various rule-based algorithms such as constraints solvers, decision procedures, theorem provers and algorithms expressed in logic programming languages, and to provide a modular framework for studying their combinations. A general overview of ELAN can be found in Borovanksy *et al.* (1998), whereas Kirchner & Ringeissen (1998) and Castro (1998) (to which we shall return in Section 11) describe applications of ELAN to constraint programming and constraint solving.

Also, in the hybrid functional and object-oriented language programming language CLAIRE (Caseau & Laburthe, 1996), rules are present. CLAIRE was designed to apply constraint programming techniques for operations research problems. The rule-based programming is supported by means of production rules that can be naturally used to express constraint propagation.

It is useful to mention here that logic programming and constraint logic programming are also rule-based formalisms. However, these formalisms use rules differently than rule-based programming described above. This distinction is usually captured by referring to *forward chaining* and *backward chaining*. In rule-based programming, as discussed above, forward chaining is used, while in logic programming and constraint logic programming backward chaining is employed. Intuitively, forward chaining aims at a simplification of the considered problem, and it maintains equivalence, while backward chaining models reasoning by cases, where each case is implicitly represented by a different rule. Both forms of chaining can be combined,

and in fact such a combination is realized in the CHR language, in which the CHR rules model forward chaining while the usual Prolog rules model backward chaining.

1.2 Overview of our approach

The traditional way of solving CSPs consists of combining constraint propagation techniques with search. Constraint propagation aims at reducing a CSP to an equivalent one but simpler. In the case of finite domains, the most basic approach to search consists of labeling, a repeated enumeration of the domains of the successive variables.

The aim of this paper is to show that constraint programming can be entirely rendered by means of rule-based programming. To this end, we provide a framework in which one computes on CSPs by means of rules represented by simple first-order formulas. In this approach, the constraint propagation is achieved by repeated application of the rules, while search is limited to labeling. This yields a framework for constraint programming more related to logic than the usual one based on algorithms achieving local consistency.

The rules we shall consider are implications built out of simple atomic formulas. In our study, we focus on two types of rules. The first type, that we call *equality rules*, are of the form

$$x_1 = s_1, \ldots, x_n = s_n \rightarrow y \neq t$$

where $x_1, \ldots, x_n, y$ are variables and $s_1, \ldots, s_n, t$ are elements of the respective variable domains. The computational interpretation of such a rule is:

if for $i \in [1..n]$ the domain of the variable x_i equals the singleton $\{s_i\}$, then remove the element t from the domain of y.

The second type of rules, that we call *membership rules*, are of the form

$$x_1 \in S_1, \ldots, x_n \in S_n \rightarrow y \neq t$$

where

- $x_1, \ldots, x_n$ are variables and $S_1, \ldots, S_n$ are subsets of the respective variable domains, and
- y is a variable and t is an element of its domain.

The computational interpretation of such a rule is:

if for $i \in [1..n]$ the domain of the variable x_i is included in the set S_i, then remove the element t from the domain of y.

To illustrate the use of these rules we study CSPs that are built out of predefined, explicitly given finite constraints. Such CSPs often arise in practice. Examples include Boolean constraints, constraints dealing with Waltz's language for describing polyhedral scenes, Allen's temporal logic, and constraints in any multi-valued logic.

To solve such CSPs we explore the structure of these explicitly given constraints first. This information is expressed in terms of valid equality and membership rules. The computation process for a CSP built out of these constraints consists of two

phases: a generation of the rules from the explicitly given constraints; and a repeated application of these rules, combined with labeling.

To characterize the effect of the generated equality and membership rules, we use the notion of local consistency. The notion approximates in a loose sense the notion of 'global consistency' (e.g. see Tsang, 1993). We show that the first type of rule leads to a local consistency notion that turns out to be weaker than arc consistency for constraints of arbitrary arity. We call it *rule consistency*.

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 are similar to the well-known Boolean propagation rules (e.g. see Frühwirth, 1998, p. 113). As a side effect, this shows that the Boolean propagation rules characterize arc consistency. Rule consistency is thus a generalization of the Boolean propagation to non-binary domains.

We also show that the membership rules lead to a notion of local consistency that coincides with arc consistency. This yields a rule-based implementation of arc consistency.

To show feasibility of this rule-based approach to constraint programming, we automatically generate both types of rules, for an explicitly given finite constraint, as rules in the CHR language. When combined with a labeling procedure such CHR, programs constitute automatically derived decision procedures for the considered CSPs, expressed on the constraint programming language level. In particular, we automatically generate the algorithms that enforce rule consistency and arc consistency.

The availability of the algorithms that enforce rule consistency and arc consistency on the constraint programming language level further contributes to the automation of the programming process within the constraint programming framework. In fact, in the case of such CSPs 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 equality rules and membership rules, we can implement more powerful notions of local consistency.

Alternatively, the equality rules and membership rules generated could be fed into any of the generic *Chaotic Iteration* algorithms of Apt (1999a), 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, in particular, 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.

1.3 Organization of the paper

The rest of the paper is organized as follows. In the next section, we clarify the syntax of the rules, and explain how one can compute with them. In Section 3, we illustrate the use of these computations by means of an example. In Section 4 we prove that the outcomes of the computations we are interested in are unique. In Section 5 we introduce some semantic aspects of the rules, and in Section 6 we formalize that the concept of a CSP is built out of predefined constraints. Next, in Section 7 we introduce 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 8 we compare rule consistency with arc consistency. In Section 9 we study membership rules, and discuss an algorithm analogous to that of Section 7. This entails a notion of local consistency that turns out to be equivalent to arc consistency.

In Section 10 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. Finally, in Section 11 we discuss other works in which a link was made between constraint programming and rule-based programming, and in Section 12 we assess the merits of our approach. In the appendix, we summarize the tests carried out by means of our implementation of both algorithms.

2 Computing with rules

In what follows, we introduce specific type of rules and explain how one can compute with them on CSPs. First, we introduce constraints.

Consider a sequence of variables $X := x_1, \ldots, x_n$ where $n \geqslant 0$, with respective domains $D_1, \ldots, D_n$ 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 $D_1 \times \ldots \times D_n$. Given an element $d := d_1, \ldots, d_n$ of $D_1 \times \ldots \times D_n$ and a subsequence $Y := x_{i_1}, \ldots, x_{i_r}$ of X we denote by $d[Y]$ the sequence $d_{i_1}, \ldots, d_{i_r}$. In particular, for a variable x_i from X, $d[x_i]$ denotes d_i.

Next, we define the rules we are interested in.

Definition 2.1

- Let x be a variable, a an element and S a set. By an *atomic formula* we mean one of the following formulas: $x = a$, $x \neq a$, $x \in S$.
- By a *rule* we mean an expression of the form $A_1, \ldots, A_m \rightarrow B_1, \ldots, B_n$, where each A_i and B_j is an atomic formula. □

In what follows, a rule will be always associated with some constraint. Then every atomic formula $x = a$ or $x \neq a$ (respectively, $x \in S$) will be such that a belongs to the domain of x (respectively, S is a subset of the domain of x).

Subsequently, we explain how to compute using the rules in presence of constraints. First, we limit our considerations to the rules of the form $A_1, \ldots, A_m \rightarrow x \neq a$. We need to explain how to turn the disequality formula into an action. This is done by identifying the disequality $x \neq a$ with the assignment $D_x := D_x - \{a\}$, where D_x is the current domain of x. In other words, we interpret $x \neq a$ as an action of removing the value a from the current domain of the variable x.

This leads us to the definition of an application of such a rule. We need some semantic notions first.

Definition 2.2
Consider a constraint C on a sequence of variables X, a variable x of X, and a tuple $d \in C$.

- Given an atomic formula A involving x we define the relation $\models_d A$ as follows:
  - $\models_d x = a$ iff $d[x] = a$,
  - $\models_d x \neq a$ iff $d[x] \neq a$,
  - $\models_d x \in S$ iff $d[x] \in S$.
- Given a sequence of atomic formulas $\mathbf{A} := A_1, \ldots, A_m$ we define $\models_d \mathbf{A}$ iff $\models_d A_i$ for all $i \in [1..m]$. □

Definition 2.3
Consider a constraint C on a finite sequence of variables X and a rule of the form $\mathbf{A} \rightarrow x_i \neq a$ involving only variables from X.

Suppose that for all $d \in C$ we have $\models_d \mathbf{A}$. Let C' be the constraint obtained from C by removing the element a from the domain D_i of the variable x_i and by removing from C all tuples d such that $d[x_i] = a$. Then we call the constraint C' the *result of applying the rule* $\mathbf{A} \rightarrow x_i \neq a$ *to* C.

If $a \in D_i$, then we say that this is a *relevant application* of the rule $\mathbf{A} \rightarrow x_i \neq a$ to C. If C' coincides with C, we say that this application of the rule $\mathbf{A} \rightarrow x_i \neq a$ to C *maintains equivalence.* □

So the application of the rule $\mathbf{A} \rightarrow x_i \neq a$ to a constraint C on the sequence $x_1, \ldots, x_n$ of variables with respective domains $D_1, \ldots, D_n$ results in the constraint C' on the variables $x_1, \ldots, x_n$ with respective domains $D_1, \ldots, D_{i-1}, D_i', D_{i+1}, \ldots, D_n$, where

- $D_i' = D_i - \{a\}$,
- $C' = C \cap (D_1 \times \ldots \times D_{i-1} \times D_i' \times D_{i+1}, \ldots, D_n)$.

We say then that the constraint C *is restricted to the domains* $D_1, \ldots, D_{i-1}, D_i', D_{i+1}, \ldots, D_n$.

Now that we defined the result of a single application of a rule we proceed to define computations. To this end we first introduce CSPs.

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

By a *solution* to $\langle \mathscr{C} \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \rangle$ we mean an element $d \in D_1 \times \ldots \times D_n$

such that for each constraint $C \in \mathscr{C}$ on a sequence of variables X we have $d[X] \in C$. We call a CSP *consistent* if it has a solution. Two CSPs with the same sequence of variables are called *equivalent* if they have the same set of solutions.

We now modify the definition of an application of a rule to a constraint to an application of a rule to a CSP. To this end, we attach each rule to a constraint to which it is supposed to be applied. Even though the constraints change during the computations we consider, it will be always clear from the context to which constraint a given rule is attached.

Definition 2.4
Consider a CSP $\mathscr{P}$ and a rule $\mathbf{A} \rightarrow x_i \neq a$ attached to a constraint C of $\mathscr{P}$. Suppose that for all $d \in C$ we have $\models_d \mathbf{A}$. Define a CSP $\mathscr{P}'$ on the same variables as $\mathscr{P}$ as follows:

- the domain of x_i in $\mathscr{P}'$ equals $D_i - \{a\}$, where D_i is the domain of x_i in $\mathscr{P}$,
- the domains of other variables in $\mathscr{P}'$ are the same as in $\mathscr{P}$,
- the constraints of $\mathscr{P}'$ are obtained by restricting the constraints of $\mathscr{P}$ to the new domains.

We say then that the CSP $\mathscr{P}'$ is the *result of applying the rule* $\mathbf{A} \rightarrow x_i \neq a$ *to* $\mathscr{P}$. If $a \in D_i$, then we say that this is a *relevant application* of the rule $\mathbf{A} \rightarrow x_i \neq a$ to $\mathscr{P}$. □

Finally, we introduce the crucial notion of a computation.

Definition 2.5
Consider a set of rules $\mathscr{R}$ of the form $\mathbf{A} \rightarrow x \neq a$ and an initial CSP $\mathscr{P}$. By a *computation by means of* $\mathscr{R}$ *starting at* $\mathscr{P}$ we mean a maximal sequence of CSPs $\mathscr{P}_1, \ldots, \mathscr{P}_i, \ldots$ such that each $\mathscr{P}_{j+1}$ is the result of a relevant application of a rule from $\mathscr{R}$ to $\mathscr{P}_j$. □

Note that when the set of rules $\mathscr{R}$ is finite or when all domains in $\mathscr{P}$ are finite, all computations starting at $\mathscr{P}$ are finite. The reason is that in each of these two cases the number of elements mentioned in the conclusions of the rules in $\mathscr{R}$ is finite. But each element can be removed from a domain only once and we insist that in computations each rule application is relevant, from which the claim follows. If a computation is finite, then no application of a rule from the considered set of rules $\mathscr{R}$ to the final CSP is relevant, i.e. this final CSP is *closed under the rules in* $\mathscr{R}$.

The computations are a means to reduce the domains of the variables while preserving the equivalence of the considered CSP. The computations here considered are in general insufficient for solving a CSP and in the case of CSPs with finite domains they have to be combined with labeling. Labeling can be modeled in the above rule-based framework by introducing a rule that splits a given CSP into two, the union of which is equivalent to the given CSP. The addition of such a rule to the considered framework leads to no conceptual difficulties and is omitted. On the other hand, various forms of labeling strategies, like the one in which variable with the smallest domain is chosen first, cannot be captured on this level.

The above string of definitions allowed us to define computations in which the

actions are limited to the applications of rules of the form $\mathbf{A} \rightarrow x \neq a$ acting on CSPs.

By limiting our attention to such type of rules we do not lose any expressiveness. Indeed, consider first a rule of the form $\mathbf{A} \rightarrow x = a$. To compute with it we interpret the equality $x = a$ as the assignment $D_x := D_x \cap \{a\}$, where D_x is the current domain of x. Then each rule $\mathbf{A} \rightarrow x = a$ is equivalent to the conjunction of the rules of the form $\mathbf{A} \rightarrow x \neq b$ with $b \in D - \{a\}$, where D is the original domain of x.

Next, consider a rule of the form $\mathbf{A} \rightarrow x \in S$. To compute with it, we interpret the atomic formula $x \in S$ as the assignment $D_x := D_x \cap S$, where D_x is the current domain of x. Then each rule $\mathbf{A} \rightarrow x \in S$ is equivalent to the conjunction of the rules of the form $\mathbf{A} \rightarrow x \neq b$ with $b \in D - S$, where D is the original domain of x.

Finally, each rule of the form $\mathbf{A} \rightarrow B_1, \ldots, B_m$ is equivalent to the conjunction of the rules of the form $\mathbf{A} \rightarrow B_i$ for $i \in [1..m]$.

Note that the rules of the form $x_1 = a_1, \ldots, x_n = a_n \rightarrow y \neq b$ are more expressive than so-called *dependency rules* of database systems (e.g. see Ullman, 1988). These are rules of the form $x_1 = a_1, \ldots, x_n = a_n \rightarrow y = b$. We just explained how to model them by means of rules of the form $x_1 = a_1, \ldots, x_n = a_n \rightarrow y \neq b$.

However, modeling in the other direction is not possible, as can be seen by taking the variables x, y, each with the domain $\{0, 1, 2\}$, and the constraint C on x, y represented by the following table:

x	y
0	1
0	0
2	2

Then the rule $x = 0 \rightarrow y \neq 2$ is not equivalent to a conjunction of the dependency rules.

3 An example

We now show how we can use the rules for computing by means of an example kindly provided to us by Victor Marek. We solve here a simple logic puzzle from Fleming (2000).

Below, given a set of variables $x_1, \ldots, x_n$, each with the domain D we denote the following set of rules:

$$\{x_i = a \rightarrow x_j \neq a \mid i, j \in [1..n], i \neq j, \ a \in D\}$$

by *all_different*$(x_1, \ldots, x_n)$. These rules formalize the requirement that the variables $x_1, \ldots, x_n$ are all different. The puzzle is as follows.

To stave off boredom on a rainy Saturday afternoon, Ms. Rojas invented a game for Denise and her two other children to play. Each child selected a different household object (no two of which were in the same room) to describe to the others, who would try to guess the item and its

location in the house. Can you match each child with the item he or she selected to describe, as well as the room of the house (one is the living room) in which each is located?

Here are the clues provided:

1. *The three children are Byron, the child who selected the book, and the one whose item is in the den;*
2. *The rug is in the dining room;*
3. *Felicia selected the picture frame.*

To solve this puzzle we use nine variables,

- $child_1, child_2, child_3$, to denote the three children, Byron, Denise and Felicia,
- $room_B, room_D, room_F$, to denote the rooms of, respectively, Byron, Denise and Felicia,
- $item_B, item_D, item_F$, to denote the objects selected by, respectively, Byron, Denise and Felicia.

We postulate that the domain of $child_1$ is $\{Byron\}$, that of $child_2$ is $\{Denise\}$ and that of $child_3$ it is $\{Felicia\}$. Next, we assume that each $room_i$ variable has the set $\{\text{den}, \text{dining}, \text{living}\}$ as its domain, and each $item_i$ variable has the set $\{\text{book}, \text{frame}, \text{rug}\}$ as its domain.

The initial set up of the story is formalized by the following rules:

- $all\_different(room_B, room_D, room_F)$,
- $all\_different(item_B, item_D, item_F)$.

This yields 36 rules, but we shall group the rules with the same premise, so we shall actually have 18 rules. The rules we shall need below will be, from the first set:

(r1) $room_B = \text{dining} \rightarrow room_D \neq \text{dining}, room_F \neq \text{dining}$,
(r2) $room_B = \text{living} \rightarrow room_D \neq \text{living}, room_F \neq \text{living}$,
(r3) $room_D = \text{living} \rightarrow room_B \neq \text{living}, room_F \neq \text{living}$,

and from the second set:

(r4) $item_B = \text{rug} \rightarrow item_D \neq \text{rug}, item_F \neq \text{rug}$,
(r5) $item_F = \text{frame} \rightarrow item_B \neq \text{frame}, item_D \neq \text{frame}$.

Next, the first clue is formalized by means of eight rules, out of which the only ones of relevance below will be

(c1.1) $\rightarrow item_B \neq \text{book}$,
(c1.2) $item_D = \text{book} \rightarrow room_D \neq \text{den}$.

The second clue is formalized by means of six rules, out of which the only one of relevance below will be

(c2) $item_B = rug \rightarrow room_B \neq \text{den}, room_B \neq \text{living}$.

Finally, the third clue is formalized by means of two rules:

(c3.1) $\rightarrow item_F \neq \text{rug}$,

(c4.1) $\rightarrow item_F \neq$ book.

So in total we have 34 rules, but we shall use below only the ten rules made explicit. The initial CSP has nine variables as introduced above and one single 'universal' constraint that consists of the Cartesian product of all the variable domains. The computation consists of 12 steps and proceeds as follows:

1. Using rule (c3.1) the domain of $item_F$ is limited to {book, frame}.
2. Using rule (c3.2) the domain of $item_F$ is further limited to {frame}. Thus $item_F$ = frame is established.
3. Using rule (r5) and the fact $item_F$ = frame just established the domain of $item_B$ is limited to {book, rug}.
4. Using rule (c1.1) the domain of $item_B$ is further limited to {rug}. Thus, $item_B$ = rug is established.
5. Using rule (r5) and the fact $item_F$ = frame established in step 2 the domain of $item_D$ is limited to {book, rug}.
6. Using rule (r4) and the conclusion $item_B$ = rug of step 4 the domain of $item_D$ is further limited to {book}. Thus, $item_D$ = book is established.
7. Using rule (c2) and the fact $item_B$ = rug established in step 4, the domain of $room_B$ is limited to {dining, living}.
8. Again using rule (c2) and the fact $item_B$ = rug established in step 4, the domain of $room_B$ is further limited to {dining}. Thus $room_B$ = dining is established.
9. Using the fact $room_B$ = dining established in step 8 and rule (r1) the domain of $room_D$ is limited to {den, living}.
10. Using the fact $item_D$ = book established in step 6 and rule (c1.2), the domain of $room_D$ is further limited to {living}. Thus, $room_D$ = living is established.
11. Using the fact $room_B$ = dining established in step 9 and rule (r2) the domain of $room_F$ is limited to {den, dining}.
12. Using the fact $room_D$ = living established in step 10 and rule (r3) the domain of $room_F$ is further limited to {den}. Thus, $room_F$ = den is established.

At this stage one can check that the resulting CSP with all singleton domains is closed under all 34 rules. This yields the solution to the puzzle represented by the following table:

child	room	item
Byron	dining	rug
Denise	living	book
Felicia	den	frame

4 Outcomes of computations

A natural question arises whether the outcome of computations using a finite set of rules is unique. The answer is positive. To prove it we need a lemma concerning iterations of inflationary and monotonic functions.

Definition 4.1
Consider a partial ordering $(D, \sqsubseteq)$ with the least element $\bot$ and a finite set of functions $F := \{f_1, \ldots, f_k\}$ on D.

- By an *iteration of* F we mean an infinite sequence of values $d_0, d_1, \ldots$ defined inductively by
$$d_0 := \bot,$$
$$d_j := f_{i_j}(d_{j-1}),$$
where each i_j is an element of $[1..k]$.
- We say that an increasing sequence $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \ldots$ of elements from D *eventually stabilizes at* d if for some $j \geqslant 0$ we have $d_i = d$ for $i \geqslant j$.
- A function f on D is called *inflationary* if $x \sqsubseteq f(x)$ for all x.
- A function f on D is called *monotonic* if $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$ for all x, y.

□

We now need the following lemma from Apt (1999b).

Lemma 4.2
Consider a partial ordering $(D, \sqsubseteq)$ with the least element $\bot$ and a finite set of monotonic functions F on D. Suppose that an iteration of F eventually stabilizes at a common fixpoint d of the functions from F. Then d is the least common fixed point of the functions from F.

It follows that all iterations of F that eventually stabilize at a common fixpoint stabilize at the same element. We now prove the desired result.

Theorem 4.3
Fix an initial CSP $\mathscr{P}$. Consider a finite set $\mathscr{R}$ of rules of the form $\mathbf{A} \rightarrow x \neq a$. Then all computations by means of $\mathscr{R}$ starting at $\mathscr{P}$ yield the same CSP.

Proof
We noted in Section 2 that all such computations are finite. Suppose that $\mathscr{P} := \langle \mathscr{C} \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \rangle$. We consider now the following partial ordering $(D, \sqsubseteq)$. The elements of D are the sequences $(E_1, \ldots, E_n)$ such that $E_i \subseteq D_i$ for $i \in [1..n]$, ordered componentwise w.r.t. the reversed subset ordering $\supseteq$. So $(D_1, \ldots, D_n)$ is the least element $\bot$ in this ordering and

$$(E_1, \ldots, E_n) \sqsubseteq (F_1, \ldots, F_n) \text{ iff } E_i \supseteq F_i \text{ for } i \in [1..n].$$

We replace in each rule each premise atom $x_i = a$ by $x_i \in \{a\}$ and $x_i \neq a$ by $x_i \in D_i - \{a\}$. Since for all $d \in D_1 \times \ldots \times D_n$ we have $\models_d x_i = a$ iff $\models_d x_i \in \{a\}$ and $\models_d x_i \neq a$ iff $\models_d x_i \in D_i - \{a\}$, it follows that the applications of the original and of the resulting rules coincide. This allows us to confine our attention to the rule each premise of which is of the form $z \in S$.

Consider now a membership rule $z_1 \in S_1, \ldots, z_m \in S_m \rightarrow y \neq a$ associated with a constraint C from $\mathscr{C}$ defined on a set of variables Y. We interpret this rule as a function on the just defined set D as follows.

First, denote by $\bar{C}$ the extension 'by padding' of C to all the variables $x_1, \ldots, x_n$,

i.e. $\bar{C} \subseteq D_1 \times \ldots \times D_n$ and $d \in \bar{C}$ iff $d[Y] \in C$. Next, given a constraint E and its variable z denote the set $\{d[z] \mid d \in E\}$ by $E[z]$. Finally, assume for simplicity that y is x_n.

The function f that corresponds to the rule $z_1 \in S_1, \ldots, z_m \in S_m \rightarrow y \neq t$ is defined as follows:

$$f(E_1, \ldots, E_n) := \begin{cases} (E_1, \ldots, E_n - \{a\}) & \text{if } (\bar{C} \cap (E_1 \times \ldots \times E_n))[z_i] \subseteq S_i \\ & \text{for } i \in [1..m], \\ (E_1, \ldots, E_n) & \text{otherwise.} \end{cases}$$

Denote the set of so defined functions by F. By definition each function $f \in F$ is inflationary and monotonic w.r.t. the componentwise reversed subset ordering $\supseteq$.

Now, there is a one-one correspondence between the common fixpoints of the functions from F at which the iterations of F eventually stabilize and the outcomes of the computations by means of $\mathscr{R}$ starting at $\mathscr{P}$. In this correspondence, a common fixpoint $(E_1, \ldots, E_n)$ is related to the CSP $\langle \mathscr{C}' \; ; \; x_1 \in E_1, \ldots, x_n \in E_n \rangle$ closed under the rules of $\mathscr{R}$, where $\mathscr{C}'$ are the constraints from $\mathscr{C}$ restricted to the domains $E_1, \ldots, E_n$. The conclusion now follows by Lemma 4.2. □

5 Semantic aspects of rules

We now introduce a number of semantic notions concerning rules.

Definition 5.1
Consider a constraint C.

- We say that the rule $\mathbf{A} \rightarrow \mathbf{B}$ is *valid for* C if for all tuples $d \in C$

$$\models_d \mathbf{A} \text{ implies } \models_d \mathbf{B}.$$

- We say that the constraint C is *closed under the rule* $\mathbf{A} \rightarrow \mathbf{B}$ if

$$(\models_d \mathbf{A} \text{ for all tuples } d \in C) \text{ implies } (\models_d \mathbf{B} \text{ for all tuples } d \in C).$$

- We say that the rule $\mathbf{A} \rightarrow \mathbf{B}$ is *feasible for* C if for some tuple $d \in C$ we have $\models_d \mathbf{A}$.
- We say that the rule $\mathbf{A} \rightarrow \mathbf{B}$ for the constraint C *extends* the rule $\mathbf{A}' \rightarrow \mathbf{B}$ if $\mathbf{A}$ contains all variables of $\mathbf{A}'$ and for all tuples $d \in C$

$$\models_d \mathbf{A} \text{ implies } \models_d \mathbf{A}'.$$

- Given a set of rules $\mathscr{R}$, we call a rule *minimal in* $\mathscr{R}$ if it is feasible and it does not properly extend a valid rule in $\mathscr{R}$. □

To illustrate them consider the following example.

Example 5.2
Take as a constraint the ternary relation that represents the conjunction $and(x, y, z)$.

It can be viewed as the following table:

x	y	z
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 the definition of an application of a rule is so designed that a link with semantics is kept in the following sense: if a rule r is valid for a constraint C, then C is closed under r. Rules that are not feasible are trivially valid. Note also that a rule that extends a valid rule is valid, as well. So validity extends 'upwards'.

Note the use of the condition '**A** contains all variables of **A**'' in the definition of the relation 'the rule $\mathbf{A} \rightarrow \mathbf{B}$ extends the rule $\mathbf{A}' \rightarrow \mathbf{B}$'. Without it we would have the following paradoxical situation. Consider the variables x, y, z, all on the domains $\{0, 1\}$, and the constraint C on x, y, z defined by $C := \{(0, 0, 0), (1, 1, 1)\}$. Then the rules $x = 0 \rightarrow z \neq 1$ and $y = 0 \rightarrow z \neq 1$ are both valid for C and for all $d \in C$ we have that $\models_d x = 0$ implies $\models_d y = 0$. So without the mentioned condition we would have that the rule $x = 0 \rightarrow z \neq 1$ extends the rule $y = 0 \rightarrow z \neq 1$, which would imply that the first rule is not minimal.

In the sequel the following observation will be useful.

Note 5.3

Consider two finite and non-empty constraints C and E such that $C \subseteq E$ and a set of rules $\mathscr{R}$. Then C is closed under all valid rules from $\mathscr{R}$ for E iff it is closed under all minimal valid rules in $\mathscr{R}$ for E.

Proof

Suppose that C is closed under all minimal valid rules in $\mathscr{R}$ for E. Take a rule r from $\mathscr{R}$ that is valid for E.

Case 1. r is feasible for E.

Then, because E is finite, r extends some minimal valid rule r' in $\mathscr{R}$ for E. But C is closed under r', so it is closed under r, as well.

Case 2. r is not feasible for E.

Then r is not feasible for C either since $C \subseteq E$. Consequently, since C is non-empty, C is closed under r. □

In what follows we confine our attention to computations involving two types of rules:

- *equality rules*: these are rules of the form $x_1 = s_1, \ldots, x_m = s_m \rightarrow y \neq a$; we abbreviate them to $X = s \rightarrow y \neq a$, where $X = x_1, \ldots, x_m$ and $s = s_1, \ldots s_m$,
- *membership rules*: these are rules of the form $x_1 \in S_1, \ldots, x_m \in S_m \rightarrow y \neq a$; we abbreviate them to $X \in S \rightarrow y \neq a$, where $X = x_1, \ldots, x_m$ and $S = S_1, \ldots S_m$.

By specializing in the last clause of Definition 5.1, defining a minimal rule, the set $\mathscr{R}$ of rules to the set of equality rules and to the set of membership rules we obtain the notions of a *minimal equality rule* and of a *minimal membership rule*. For equality and membership rules the following straightforward characterization of the 'extends' relation will be of use.

Note 5.4

(i) An equality rule $x_1 = s_1, \ldots, x_m = s_m \rightarrow y \neq a$ extends an equality rule $z_1 = t_1, \ldots, z_n = t_n \rightarrow y \neq a$ iff $z_1 = t_1, \ldots, z_n = t_n$ is a subsequence of $x_1 = s_1, \ldots, x_m = s_m$.

(ii) A membership rule $x_1 \in S_1, \ldots, x_m \in S_m \rightarrow y \neq a$ extends a membership rule $z_1 \in T_1, \ldots, z_n \in T_n \rightarrow y \neq a$ iff $z_1, \ldots, z_n$ is a subsequence of $x_1, \ldots, x_m$ and for each $i \in [1..n]$ we have $S_{\pi(i)} \subseteq T_i$, where z_i equals $x_{\pi(i)}$. □

Given a CSP with finite domains we would like to solve it by considering computations starting at it. But where do we get the rules from? Note that given a constraint C and a rule r that is valid for C, the constraint C is trivially closed under r. Consequently, an application of r to C is not relevant, i.e. it does not affect C. So to obtain some change we need to use rules that are not valid for the initial constraints. This brings us to the notion of a CSP based on another one.

6 CSPs built out of predefined constraints

In the introduction we informally referred to the notion of a CSP 'being built out of predefined, explicitly given 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 6.1

- 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

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

  and call it *a permutation of* 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 6.2
We assume that the 'predefined constraints' are presented as a given in advance CSP $\mathscr{BASE}$. Suppose that each constraint C of a CSP $\mathscr{P}$ is based on a permutation of a constraint of $\mathscr{BASE}$. We say then that $\mathscr{P}$ is *based on* $\mathscr{BASE}$. □

In the above definition, the use of permutations of constraints allows us to abstract from the variable ordering used in $\mathscr{BASE}$. The following example illustrates this notion.

Example 6.3
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:

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

$\mathscr{BOOL}$ 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 $\mathscr{BOOL}$. For example, in Section 10 we shall consider the query $add(1, x, y, z, 0)$. It corresponds to the following CSP based on $\mathscr{BOOL}$:

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

□

In what follows, we consider computations that start with a CSP based on some CSP $\mathscr{BASE}$. In these computations, we wish to maintain equivalence between the successive CSPs. To this end, the following simple observation is crucial.

Note 6.4
Consider two constraints C and E such that C is based on E. Let $\mathbf{A} \rightarrow x \neq a$ be a rule valid for E. Then the application of $\mathbf{A} \rightarrow x \neq a$ to C maintains equivalence.

Proof
Assume that the rule $\mathbf{A} \rightarrow x \neq a$ can be applied to C, i.e. that for all $d \in C$ we have $\models_d \mathbf{A}$. Suppose now that the rule $\mathbf{A} \rightarrow x \neq a$ to C does not maintain equivalence. Then for some $d \in C$ we have $d[x] = a$. C is based on E, so $d \in E$. By the validity of the rule for E we get $d[x] \neq a$. This yields a contradiction. □

This observation provides us with a way of maintaining equivalence during a computation: it suffices to use at each step a rule that is valid for a permutation C^π of a constraint of $\mathscr{BASE}$. Such a rule is then attached (i.e. applied) to the constraint based on C^π. This is what we shall do in the sequel. Depending on the type of rules used we obtain in this way different notions of local consistency.

7 Rule consistency

In this section we consider a CSP $\mathscr{P}$ based on some finite CSP $\mathscr{BASE}$, and study computations that use exclusively equality rules. The rules are obtained from the constraints of $\mathscr{BASE}$; each of them is valid for a permutation C^π of a constraint C of $\mathscr{BASE}$ and is applied to the constraint of $\mathscr{P}$ based on C^π. By Note 6.4 the successive CSP's are all equivalent to the initial CSP $\mathscr{P}$. The computation ends when a CSP is obtained that is closed under the rules used. This brings us to a natural notion of local consistency expressed in terms of equality rules.

Definition 7.1
Consider a CSP $\mathscr{P}$ based on a CSP $\mathscr{BASE}$. Let C be a constraint of $\mathscr{P}$. For some constraint $f(C)$ of $\mathscr{BASE}$ and a permutation π, C is based on $f(C)^\pi$.

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

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

Example 7.2
Take as the base CSP

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

and consider the following four CSPs 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 5.2 that the equality rule $x = 0 \rightarrow z \neq 1$ is valid for $and(x,y,z)$. In the first three CSPs its only constraint is closed under this rule, while in the fourth it is not closed 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 CSPs are rule consistent, while the third one is not rule consistent, since it is not closed under the valid equality rule $z = 1 \rightarrow x \neq 0$.

When trying to generate all valid equality rules Note 5.3 allows us to confine our attention to the minimal valid equality rules. We now introduce an algorithm that given a finite constraint generates the set of all minimal valid equality 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 **insert**(r, L).

By an *assignment* to a sequence of variables X, we mean here an element s from the Cartesian product of the domains of variables 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.

Equality 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)
               END
            END
         END
      END
   END
END
```

The test that one equality rule does not extend another can be easily implemented by means of Note 5.4.i.

The following result establishes correctness of this algorithm.

Theorem 7.3

Given a constraint C the Equality Rules Generation algorithm produces in L the set of all minimal valid equality rules for C.

Proof

First note that in the algorithm all possible feasible equality rules are considered, and in the list L only the valid equality rules are retained. Additionally, a valid equality rule is retained only if it does not extend a rule already present in L.

Finally, the equality rules are considered in the order according to which those that use less variables are considered first. By virtue of Note 5.4.i, this implies that if a rule r_2 extends a rule r_1, then r_1 is considered first. As a consequence, precisely all minimal valid equality rules are retained in L. □

The above algorithm is completely straightforward, and consequently inefficient. It is easy to see that given a constraint defined over n variables, $O(n \cdot 2^n \cdot d^2)$ rules are considered in it, where d is the size of the largest variable domain. This shows that in practice this algorithm is impractical for large domains and for constraints with many variables. By representing the rules explicitly one could improve the running time of this algorithm, trading time for space. Then the test that one rule does not extend another could be eliminated from the algorithm by representing explicitly

the partial ordering defined by the relation 'equality rule r_1 extends equality rule r_2'. Each time an equality rule that is valid for C would be found, all the rules that extend it would be now disregarded in further considerations. This would reduce the number of rules considered and improve the average running time. However, it is difficult to quantify the gain obtained, and in the worst case, all the rules would still have to be considered.

In Section 10 and the appendix, we present some empirical results showing when the Equality Rules Generation becomes infeasible.

8 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 Mackworth (1977a) for binary relations, and was extended to arbitrary relations in Mohr & Masini (1988). Let us recall the definition.

Definition 8.1

- 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 8.2
Consider a CSP $\mathscr{P}$ based on a CSP $\mathscr{BASE}$. If $\mathscr{P}$ is arc consistent, then it is rule consistent w.r.t. $\mathscr{BASE}$.

Proof
Assume that $\mathscr{P}$ is arc consistent. Choose a constraint C of $\mathscr{P}$ and consider an equality rule $X = s \rightarrow y \neq a$ that is valid for $f(C)^\pi$, where f and π are as in Definition 7.1.

Suppose by contradiction that C is not closed under this rule. So for $X := x_1, \ldots, x_k$ and $s := s_1, \ldots, s_k$ the domain of each variable x_j in $\mathscr{P}$ equals $\{s_j\}$ and moreover $a \in D$, where D is the domain of the variable y in $\mathscr{P}$.

By the arc consistency of $\mathscr{P}$ there exists $d \in C$ such that $d[y] = a$. Because of the form of the domains of the variables in X, also $d[X] = s$ holds. Additionally, because $\mathscr{P}$ is based on $\mathscr{BASE}$, we have $d \in f(C)^\pi$. But by assumption, the equality rule $X = s \rightarrow y \neq a$ is valid for $f(C)^\pi$, so $d[y] \neq a$. A contradiction. □

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

Example 8.3
Take as the base the following CSP

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

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 table:

x	y
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 $\mathscr{BASE}$ 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 equality rules that are valid for C. □

We now show that 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 8.4

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

Proof

The ($\Rightarrow$) implication is the contents of Theorem 8.2.

To prove the reverse implication suppose that some constraint C of $\mathscr{P}$ is not arc consistent. We prove that then C is not rule consistent.

The constraint C is on some variables $x_1,\ldots,x_n$ with respective domains $D_1,\ldots,D_n$. For some $i \in [1..n]$ some $a \in D_i$ does not participate in any solution to C.

Let $D_{i_1},\ldots,D_{i_\ell}$ be the sequence of all domains in $D_1,\ldots,D_{i-1},D_{i+1},\ldots,D_n$ that are singletons. Suppose that $D_{i_j} := \{s_{i_j}\}$ for $j \in [1..\ell]$ and let $X := x_{i_1},\ldots,x_{i_\ell}$ and $s := s_{i_1},\ldots,s_{i_\ell}$.

Consider now the equality rule $X = s \rightarrow x_i \neq a$ and take $f(C)^\pi$, where f and π are as in Definition 7.1. For appropriate domains $D'_1,\ldots,D'_n$ of $\mathscr{BASE}$ we have $f(C)^\pi \subseteq D'_1 \times \ldots \times D'_n$.

Next, take some $d \in f(C)^\pi$ such that $d[X] = s$. We show that $d \in C$. Since $C = f(C)^\pi \cap (D_1 \times \ldots \times D_n)$ it suffices to prove that $d \in D_1 \times \ldots \times D_n$. For each variable x_j lying inside of X we have $d[x_j] = s_j \in D_j$. In turn, for each variable x_j lying outside of X its domain D_j has two elements, so, by the assumption on $\mathscr{BASE}$, D_j is the same as the corresponding domain D'_j of $f(C)^\pi$ and consequently $d[x_j] \in D_j$, since $d \in D'_1 \times \ldots \times D'_n$.

So indeed, $d \in C$, and hence $d[x_i] \neq a$ by the choice of a. This proves validity of the equality rule $X = s \rightarrow x_i \neq a$ for $f(C)^\pi$.

But C is not closed under this rule, since $a \in D_i$, so C is not rule consistent. □

9 Membership rule consistency

In this section, we consider computations that use exclusively membership rules. In the previous section we saw that the notion of rule consistency is weaker than that of arc consistency for constraints of arbitrary arity. Here we show that by using the membership rules, we obtain a notion of local consistency that coincides with arc consistency.

First, let us clarify the notion of a membership rule by considering the following example.

Example 9.1
Consider a constraint on variables x, y, z, each with the domain $\{+, -, l, r\}$, that is defined by the following table:

x	y	z
$+$	$+$	$+$
$-$	$-$	$-$
l	r	$-$
$-$	l	r
r	$-$	l

This constraint is the so-called *fork* junction in the language of Waltz (1975) for describing polyhedral scenes. Note that the following three membership rules

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

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

and

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

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

Now, in analogy to Definition 7.1, we introduce the following notion.

Definition 9.2
Consider a CSP $\mathscr{P}$ is based on a CSP $\mathscr{BASE}$. Let C be a constraint of $\mathscr{P}$. For some constraint $f(C)$ of $\mathscr{BASE}$ and a permutation π, C is based on $f(C)^\pi$.

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

We now have the following result.

Theorem 9.3
Consider a CSP $\mathscr{P}$ based on a CSP $\mathscr{BASE}$. Then $\mathscr{P}$ is arc consistent iff it is membership rule consistent w.r.t. $\mathscr{BASE}$.

Proof
($\Rightarrow$) This part of the proof is a simple modification of the proof of Theorem 8.2.

Assume that $\mathscr{P}$ is arc consistent. Choose a constraint C of $\mathscr{P}$ and consider a membership rule $X \in S \rightarrow y \neq a$ that is valid for $f(C)^\pi$, where f and π are as in Definition 6.2.

Suppose by contradiction that C is not closed under this rule. So for $X := x_1, \ldots, x_k$ and $S := S_1, \ldots, S_k$ the domain of each variable x_j is included in S_j and moreover $a \in D$, where D is the domain of the variable y.

By the arc consistency of $\mathscr{P}$ there exists $d \in C$ such that $d[y] = a$. Because of the form of the domains of the variables in X, also $d[x_i] \in S_i$ for $i \in [1..k]$ holds. Additionally, because $\mathscr{P}$ is based on $\mathscr{BASE}$ we have $d \in f(C)^\pi$. But by assumption the rule $X \in S \rightarrow y \neq a$ is valid for $f(C)^\pi$, so $d[y] \neq a$. A contradiction.

($\Leftarrow$) This part of the proof is a modification of the proof of Theorem 8.4.

Suppose that some constraint C of $\mathscr{P}$ is not arc consistent. We prove that then C is not membership rule consistent. The constraint C is on some variables $x_1, \ldots, x_n$ with respective domains $D_1, \ldots, D_n$. For some $i \in [1..n]$ some $a \in D_i$ does not participate in any solution to C.

Take $f(C)^\pi$, where f and π are as in Definition 9.2. For appropriate domains $D'_1, \ldots, D'_n$ of $\mathscr{BASE}$ we have $f(C)^\pi \subseteq D'_1 \times \ldots \times D'_n$.

Let $D_{i_1}, \ldots, D_{i_\ell}$ be the sequence of domains in $D_1, \ldots, D_{i-1}, D_{i+1}, \ldots, D_n$ that are respectively different than $D'_1, \ldots, D'_{i-1}, D'_{i+1}, \ldots, D'_n$. Further, let $X := x_{i_1}, \ldots, x_{i_\ell}$ and $S := D_{i_1}, \ldots, D_{i_\ell}$.

Consider now the membership rule $X \in S \rightarrow x_i \neq a$. Take some $d \in f(C)^\pi$ such that $d[x_{i_j}] \in D_{i_j}$ for $j \in [1..\ell]$. We show that $d \in C$. Since $C = f(C)^\pi \cap (D_1 \times \ldots \times D_n)$ it suffices to prove that $d \in D_1 \times \ldots \times D_n$. For each variable x_j lying inside of X we have $d[x_j] \in D_j$. In turn, for each variable x_j lying outside of X its domain D_j is the same as the corresponding domain D'_j of $f(C)^\pi$ in $\mathscr{BASE}$ and consequently $d[x_j] \in D_j$, since $d \in D'_1 \times \ldots \times D'_n$.

So indeed, $d \in C$, and hence $d[x_i] \neq a$ by the choice of a. This proves validity of the rule $X \in S \rightarrow x_i \neq a$ for $f(C)^\pi$. But C is not closed under this membership rule since $a \in D_i$, so C is not membership rule consistent. □

Example 8.3 shows that the notions of rule consistency and membership rule consistency do not coincide. To see this difference better, let us reconsider the CSP discussed in this example.

We noted there that this CSP is not arc consistent, and that it is rule consistent. From the above theorem, we know that this CSP is not membership rule consistent. In fact, consider the following membership rule:

$$x \in \{0, 1\} \rightarrow y \neq 2.$$

This membership rule is valid for the base constraint C, but the restricted constraint $C \cap (D_1 \times D_2)$ is not closed under this rule. In conclusion, the membership rules are more powerful than the equality rules.

As in Section 7, we now provide an algorithm that, given a constraint, generates the set of all minimal valid membership 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 Equality Rules Generation algorithm, we now need a slightly different notion. To define it for each variable x from VAR recall that we denoted 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.

Membership 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)
               END
            END
         END
      END
   END
END
```

The test that one membership rule does not extend another can be implemented using Note 5.4.ii.

The following result establishes correctness of this algorithm.

Theorem 9.4
Given a constraint C the Membership Rules Generation algorithm produces in L the set of all minimal valid membership rules for C.

Proof
The proof is analogous to that of Theorem 7.3. We only need to check that the membership rules are considered in such an order that if a rule r_2 extends a rule r_1, then r_1 is considered first. This follows from directly from Note 5.4.ii. □

10 Applications

In this section, we discuss the implementation of the Equality Rules Generation and Membership Rules Generation algorithms and discuss their use on selected domains.

*10.1 Constraint handling rules (*CHR*)*

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 Membership Rules Generation algorithm. We made a compromise between memory usage and performance so that we could tackle some non-trivial problems (in terms of size of the domains of variables, and in terms of arity of constraints) in spite of the exponential complexity of the algorithms. These implementations generate CHR rules that deal with finite domain variables using an ECL^{i}PS^e library.

Constraint Handling Rules (CHR) of Frühwirth (1995) 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.

A CHR program is a finite set of CHR rules. These rules are basically of two types (there is a third type of rules which is a combination of the first two types): simplification rules and propagation rules. When all guards are satisfied, a simplification rule replaces constraints by simpler ones while preserving logical equivalence, whereas a propagation rule adds logically redundant constraints. More precisely, these rules have the following form:

$$\begin{array}{ll} \text{simplification} & H_1,\ldots,H_i <=> G_1,\ldots,G_j \mid B_1,\ldots,B_k \\ \text{propagation} & H_1,\ldots,H_i ==> G_1,\ldots,G_j \mid B_1,\ldots,B_k \end{array}$$

where

- $i > 0$, $j \geqslant 0$, $k \geqslant 0$,
- the multi-head $H_1,\ldots,H_i$ is a non-empty sequence of CHR constraints,
- the guard $G_1,\ldots,G_j$ is a sequence of built-in constraints,
- the body $B_1,\ldots,B_k$ is a sequence of built-in and CHR constraints.

Our equality rules and membership rules can be modeled 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 unifies with `cons(0,B,1)`, this rule is fired and the value 2 is removed from the domain of the variable `B`.

In turn, the Membership Rules Generation algorithm generates rules such as $(A, C) \in (\{0\}, \{1,2\}) \rightarrow B \neq 2$. 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 unifies with 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.

10.2 Generating the rules

We begin by discussing the generation of equality rules and membership 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 5.2. The Equality Rules Generation algorithm generated in 0.02 seconds the following six equality 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 rules are somewhat different than the well-known rules that can be found, for example, in Frühwirth (1998, p. 113), where instead of the rules

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

the rules

```
and(1,X,Y) ==> X = Y.
and(X,1,Y) ==> X = Y.
and(X,Y,Z), X = Y ==> Y = Z.
```

appear. We shall discuss this matter in Section 11.

In this case, by virtue of Theorem 8.4, the notions of rule and arc consistency coincide, so the above six equality rules characterize the arc consistency of the and constraint. Our implementations of the Equality Rules Generation and the Membership Rules Generation algorithms yield here the same rules.

Three valued logic Next, consider the three valued logic of Kleene (1952, p. 334) that consists of three values, t (true), f (false) and u (unknown). We only consider here the crucial equivalence relation ≡ defined by the truth table

≡	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 equality rules and 26 membership rules. Typical examples are

```
equiv(X,Y,f) ==> X##u,Y##u.
```

and

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

Six valued logic In Van Hentenryck *et al.* (1992), the constraint logic programming language CHIP is used for the Automatic Test-Pattern Generation (ATPG) for the digital circuits. To this end, the authors define a specific six valued logic, and provide some rules (expressed in the form of so-called demons) to carry out the constraint propagation. The and6 constraint in question is defined by means of the following table:

and6	0	1	d	dnot	e	enot
0	0	0	—	—	0	0
1	0	1	d	dnot	e	enot
d	—	d	—	—	d	—
dnot	—	dnot	—	—	—	dnot
e	0	e	d	—	e	0
enot	0	enot	—	dnot	0	enot

The Equality Rules Generation algorithm generated 41 equality rules in 0.15 seconds, while the Membership Rules Generation algorithm generated 155 membership rules in 14.35 seconds. The generated rules enforce, respectively, rule consistency and arc consistency, while it is not clear what notion of local consistency is enforced by the (valid) rules of Van Hentenryck *et al.* (1992, p. 133) because some of the latter ones allow equalities between the variables in the premise. This makes the

comparison in terms of strength of the entailed notion of local consistency difficult. It is clear that our approach is more systematic and fully automatic. (In fact, we found two typo's in the rules of Van Hentenryck *et al.* (1992, p. 133).)

Propagating signs As a next example consider the rules for propagating signs in arithmetic expressions (e.g. see Davis, 1987, p. 303). We limit ourselves to the case of multiplication. Consider the following table:

×	neg	zero	pos	unk
neg	pos	zero	neg	unk
zero	zero	zero	zero	zero
pos	neg	zero	pos	unk
unk	unk	zero	unk	unk

This table determines a ternary constraint `msign` that consists of 16 triples, for instance (neg, neg, pos) that denotes the fact that the multiplication of two negative numbers yields a positive number. The value 'unk' stands for 'unknown'. The Equality Rules Generation algorithm generated in 0.08 seconds 34 equality rules. A typical example is `msign(X,zero,Y) ==> Y##pos,Y##neg,Y##unk`.

In turn, the Membership Rules Generation algorithm generated in 0.6 seconds 54 membership rules. A typical example is

```
msign(X,unk,Y) ==> in(Y,[neg, pos, zero]) | X##pos,X##neg
```

that corresponds to the following two membership rules for the constraint `msign(X,Z,Y)`

$$(Z, Y) \in (\{\mathtt{unk}\}, \{\mathtt{neg}, \mathtt{pos}, \mathtt{zero}\}) \rightarrow X \neq \mathtt{pos}$$

and

$$(Z, Y) \in (\{\mathtt{unk}\}, \{\mathtt{neg}, \mathtt{pos}, \mathtt{zero}\}) \rightarrow X \neq \mathtt{neg}.$$

Waltz' language for describing polyhedral scenes Waltz' language consists of four constraints. One of them, the fork junction was already mentioned in Example 9.1. The Equality Rules Generation algorithm generated 12 equality rules for it, and the Membership Rules Generation algorithm generated 24 membership rules.

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

x	y	z
r	l	+
r	l	−
r	l	r
r	l	l

In this case the Equality Rules Generation algorithm and the Membership 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 CSPs based on the T junction.

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

10.3 Using the rules

Next, we show by means of some examples how the generated rules can be used to reduce or 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 polyhedral scenes The following predicate describes the impossible scene given in Figure 1 and taken from Winston (1992, p. 262):

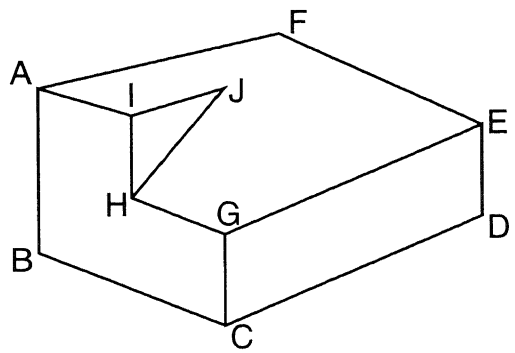

Fig. 1. Impossible scene.

```
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 table:

x	y
+	+
−	−
l	r
r	l

Here and elsewhere we use the ECL^{i}PS^e built-in `::` to declare variable domains.

When using the equality rules obtained by the Equality 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 membership 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 Winston (1992, p. 260), the membership rules are also more powerful than the equality rules, and both sets of rules reduce the problem, but in both cases labeling is needed to produce all four solutions.

Comparing the constraint solver based on the membership rules to the constraint solver based on the equality rules is not easy: although propagation is more efficient with the membership rules, the solver based on the equality rules can sometimes be faster, depending on the structure of the problem and on whether the labeling is needed.

We also compared the solvers generated by the implementations of the Equality Rules Generation and Membership Rules Generation algorithms to the approach described in By (1997) and based on meta-programming in Prolog. We ran the same examples and drew the following conclusions. For small examples our solvers were less efficient than those of By (with factors varying from 2 to 10). However, for more complex examples, our solvers became significantly more efficient, with factors varying from 10 to 500. This can be attributed to the increased role of the constraint propagation that reduces backtracking and that is absent in By's approach.

Temporal reasoning In Allen's (1983) approach to temporal reasoning, the entities are intervals and the relations are temporal binary relations between them. Allen (1983) 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, Allen (1983) provided a 13 × 13 table. This table determines a ternary constraint between a triple of events, A, B and C that we denote by allen. For example,

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

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

Using this table, the Equality Rules Generation algorithm produced for the constraint allen 498 equality rules in 31.16 seconds. In contrast, we were unable to generate all membership rules in less than 24 hours. This shows the limitations of our implementation. We tried the generated set of equality rules to solve the following problem from Allen (1983): '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 ⟨allen ; R1 ∈ [o—,m—],R2 ∈ [b,m,b—,m—],R3 ∈ TEMP⟩, where allen 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],
allen(R1,R2,R3),
labeling([R1,R2,R3]).
```

We then obtain the following solutions in 0.06 seconds:
(R1,R2,R3) ∈ {(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 Allen, 1983), 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 ∈ [o,s,d].

We now run the following query:

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

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

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

Three valued logic Next, consider the `and3` constraint in the three valued logic of Kleene (1952, p. 334) represented by the truth table

and3	t	f	u
t	t	f	u
f	f	f	f
u	u	f	u

Typical examples of the 16 generated equality rules and of the 18 generated membership rules are:

```
and3(u,u,X) ==> X##0.
```

and

```
and3(X,Y,Z) ==> in(X,[0, u]) | Z##1.
```

Consider now the query:

```
[X,Y,Z,T,U]::[0,1,u], and3(X,Y,Z), and3(T,U,Z), Z##0,Y##u, X##u.
```

Using the membership rules, we reach the answer in the form of a complete assignment to all the variables, namely `X = 1,Y = 1,Z = 1,T = 1,U = 1`, whereas using only the equality rules we do not obtain any reduction and labeling is needed to produce the same answer.

Full adder It is often the case that dealing with a compound constraint directly yields a stronger notion of local consistency than when dealing with each of its constituents separately. A prototypical example is the well-known `alldifferent` constraint on n variables that can be decomposed into $\frac{n\cdot(n-1)}{2}$ binary disequality constraints. Then arc consistency enforced on `alldifferent` is stronger than arc consistency enforced on each of the disequality constraints separately.

The same phenomenon arises for rule consistency. We illustrate it by means of the already discussed in Example 6.3 full adder circuit. It can be defined by the following constraint logic program (e.g. see Frühwirth, 1998) 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 the equality 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 Equality Rules Generation algorithm generated 52 equality rules in 0.27 seconds. The constraint propagation carried out by means of these equality rules is more powerful than the one carried out by means of the equality 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.

So rule consistency for a compound constraint defined by means of the basic constraints is indeed in general stronger than the rule consistency for the basic constraints treated separately. In fact, in the above case the equality rules for the `full_adder` constraint yield the relational (1,5)-consistency notion of Dechter & van Beek (1997), whereas by virtue of Theorem 8.4, the equality rules for the `and`, `or` and `xor` constraints yield a weaker notion of arc consistency.

11 Related work

11.1 Relation between constraint programming and rule-based programming

In a number of papers, a link was made between constraint programming and rule-based programming. To start with, in Montanari & Rossi (1991) a general study of constraint propagation was undertaken by defining the notion of a relaxation rule and by proposing a general relaxation algorithm that implements constraint propagation by means of a repeated application of the relaxation rules. However, this abstract view of constraint programming cannot be realized in a simple way since the application of a relaxation rule is a complex process.

In Apt (1998), we showed how constraint programming can be couched in proof theoretic terms by viewing the programming process as the task of proving the original CSP. In the proposed framework, two types of rules were proposed: deterministic ones and the splitting ones. Further, the deterministic rules were either concerned with domain reduction or constraint reduction. In the former case, the rules were called domain reduction rules, and in the latter case constraint reduction rules. Such rules are high-level abstractions and on the implementation level they can involve complex computations.

It is useful to see that the rule-based approach to constraint programming proposed in this paper is an instance of this proof theoretic view of constraint programming. Namely, the equality rules and the membership rules are examples of the domain reduction rules while labeling, the formal treatment of which is omitted here, is an example of a splitting rule.

The important gain is that the implementation of the considered here equality rules and membership rules boils down to a straightforward translation of them into the CHR syntax. This leads to an implementation of this approach to constraint programming by means of constraint logic programming. The important limitation is that this approach applies only to the CSP's built out of predefined, explicitly given finite constraints.

A similar approach to constraint programming to that of Apt (1998) was proposed in Castro (1998). In his approach the proof rules are represented as rewrite rules in the already mentioned in the introduction programming language ELAN. The rules use a richer syntax than here considered by referring to arbitrary constraints and to expressions of the form $x \in D$, where D is the current domain of the variable x. In particular, no constraint specific rules were considered. Instead, the emphasis was on showing how the general techniques of constraint programming, in particular various search strategies, can be expressed in the form of rules.

11.2 Generation of rules

Let us turn now to an overview of the recent work on rules generation. Building upon the work presented in Apt & Monfroy (1999), two articles appeared in which algorithms were presented that aim at improving the expressivity of the rules and at a more economic representation.

In Ringeissen & Monfroy (2000), rules similar to equality rules were considered. The most significant improvement is the use of parameters (i.e. unspecified constants) that leads to a decrease in the number of generated rules. Parameters are also a means for deducing equalities of variables on the right-hand side of rules. For instance, consider the following two rules with parameters a_1 and a_2 for a constraint C taken from Ringeissen & Monfroy (2000):

$$x_1 = a_1 \rightarrow x_2 = a_1 \wedge x_3 = 1 \tag{1}$$

$$x_2 = a_2 \rightarrow x_1 = a_2 \wedge x_3 = 1 \tag{2}$$

Rule (1) means that whatever the value of x_1 is, x_2 is equal to x_1 and x_3 is equal to 1. From rules (1) and (2), an equality between the variables on the right-hand side of the rule can be deduced (note that the resulting rule can always be applied):

$$\rightarrow x_2 = x_1 \wedge x_3 = 1$$

To generate such rules with parameters, Ringeissen & Monfroy (2000) combine unification in finite algebra with a rule generation algorithm. The size of the generated set of rules significantly depends on an ordering on variables. It still needs to be clarified what is the counterpart of the notion of a minimal rule in this framework, and whether the generated rules with parameters enforce rule-consistency.

Sets of rules generated in Abdennadher & Rigotti (2000) are even more compact and more expressive: multiple occurrences of variables and conjunction of constraints with shared variables are allowed on the left-hand side of rules. Moreover,

the user has the possibility to specify the admissible syntactic form of the rules: more specifically, right-hand sides of rules can consist of more complex constraints than (dis)-equality constraints. Here are two examples of rules (taken from two different sets of rules of Abdennadher & Rigotti, 2000):

$$and(x, x, z) \rightarrow x = z. \tag{3}$$

$$and(x, y, z), neg(x, y) \rightarrow z = 0. \tag{4}$$

In rule (3), equality between variables is deduced using a double occurrence of the variable x in the head, and rule (4) defines interaction between two constraints, *and* and *neg*.

Abdennadher & Rigotti (2000) also investigated what form of local consistency is enforced by the rules generated by their algorithms. When using the given constraint together with equality (between a variable and a value) on the left-hand side, and only disequality (between a variable and a value) on the right-hand side of the rules, the generated rules enforce rule consistency. However, in general, the enforced local consistency is stronger than rule consistency. In particular, it is plausible that membership rule consistency (i.e. arc consistency) is enforced when disequality constraints are allowed both on the right- and left-hand sides of rules.

In Apt (2000), two sets of rules for Boolean constraints were compared. One of them is the one presented in Section 10.2 and the other the already mentioned set of rules from Frühwirth (1998, p. 113) (with one difference irrelevant for the subsequent discussion). While both sets of rules enforce the arc consistency, it turns out that they are not equivalent. In fact, if a Boolean CSP with non-empty domains is closed under the rules from the second set, then it is closed under the rules from the first set. The converse does not hold since the CSP $\langle x \wedge y = z \ ; \ x \in \{1\}, y \in \{0,1\}, z \in \{0,1\}\rangle$ is closed under the first set of rules, but not under the second one, since it is not closed under the rule `and(1,X,Y) ==> X = Y`.

11.3 Local consistency notions

In this paper, we considered local consistency by focusing on individual, arbitrary, non-binary, constraints. In the literature, algorithms for achieving such notions of local consistency usually concentrated on achieving arc consistency of non-binary CSPs. Among them the algorithm `CN` of Mackworth (1977b), and `GAC4` of Mohr & Masini (1988) are respectively based on ideas similar to the AC-3 and AC-4 algorithms for binary constraints. However, in practice `CN` is usable only for ternary constraints and small domains, and has a large worst-case time complexity. On the other hand, the large space complexity of `GAC4` makes it usable only for constraints of small size.

The `GAC-schema` of Bessière & Régin (1997) was designed to enforce arc consistency on non-binary constraints while keeping a reasonable time and space complexity. It is based on an AC-7 like schema and allows constraints to be given explicitly, either in a positive way as in our case, or in a negative way, i.e. in the form of 'forbidden' tuples, or implicitly in the form of predicates. In order to make our framework as general as the `GAC-schema`, we could think of generating the

allowed tuples by testing all possible tuples. However, this would almost always be impractical because of space considerations.

In Bessière (1999), the following opinion was voiced on non-binary constraints: "Perhaps we should accept the idea that the constraint solving tool of the next years will apply different levels of local consistency on different constraints at each node of the search tree, ...". Our framework is amenable to such a view since we can generate rules for enforcing various types of local consistencies. For example, we can generate some equality rules for some constraints and some membership rules for other constraints, and then apply only the resulting set of rules.

Another consideration is that our constraint solving process consists of two separate phases: first a generation of the rules (a sort of compilation of the truth table of a constraint) which is done once and for all, followed by the application of the rules. Thus, the set of generated rules can be modified during the application phase, for example by combining some rules (for a more efficient application of the rules), by removing some rules or by strengthening some conditions to weaken the domain reduction process.

While preparing this revised version of the paper, we noted that a similar notion to our rule consistency notion was introduced in the context of the theory of fuzzy sets (see Pedrycz & Gomide, 1998, pp. 252–261). The notion there considered deals with rules of the form 'if x is A, then y is B', where A and B are fuzzy sets. In spite of the same name used (namely, rule consistency), the uses of both notions are different. In our case, we employ it to reduce a specific CSP to a smaller one that is rule consistent, but can be inconsistent, while in the case of the fuzzy set theory the corresponding notion is used to detect conditions for 'potential inconsistency' that arises when the rules express contradictory knowledge.

Finally, let us mention that rule generation appears in other areas of computer science. Typical examples are: programs for machine learning that construct a model of the knowledge using decision trees and production rules (e.g. see Quinlan, 1993) inductive logic programming, a logic-based approach to machine learning where logic programming rules are inferred from positive and negative examples and a background knowledge (e.g. Muggleton & de Raedt, 1994), and data mining that aims at extracting high-level representations in the form of patterns and models from data (e.g. see Agrawal *et al.* (1996), where so-called association rules are generated.).

12 Conclusions

The aim of this paper was to provide a framework in which constraint programming can be entirely reduced to rule-based programming. It involved constraint satisfaction problems built out of explicitly given constraints. In the case where the latter constraints are defined over small finite domains, these CSPs can often be solved by means of automatically generated constraint propagation algorithms.

We argued that such CSPs often arise in practice, and consequently the methods developed here can be of practical use. We believe that the approach of this paper could be applied to a study of various decision problems concerning specific multi-valued logics, and this in turn could be used for an analysis of digital circuits (e.g.

see Muth (1976), where a nine valued logic is used). Other applications could involve non-linear constraints over small finite domains and the analysis of polyhedral scenes in presence of shadows (see Waltz, 1975).

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

In this paper, we focused on systematic and automated aspects of rule-based constraint solvers. At present stage, it is difficult to compare the performance of our method (based on rule generation and subsequent rule application) with other methods based on classical constraint propagation algorithms. The reason is that our approach is currently implemented by means of CHR rules that are applied on top of Prolog, while the built-in constraint propagation algorithms are usually implemented at a lower level.

Finally, the notions of rule consistency and membership 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 Equality Rules Generation algorithm and the Membership 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 equality rules or the membership 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 all three referees for several useful suggestions. Victor Marek provided material for Section 3 and suggested Section 4.

Appendix

Table 1 illustrates the generation of the equality rules and the membership rules for various natural constraints. The first column gives the name of the constraint, the second its arity, the third the cardinality of the domains of variables, and the fourth the cardinality of the constraint. The subsequent two columns show the outcome of the Equality Rule Generation algorithm: first the number of equality rules generated, and then the computation time in seconds; the last two columns provide this information for the membership rules.

Table 1. *Generation of equality and membership rules*

Constraint	Arity	Domain size	Cardinality	Equality rules	Gen. (in s.)	Member. rules	Gen. (in s.)
fork	3	4	5	12	0.05	24	0.65
t	3	4	4	1	0.02	1	0.07
not	2	2	2	4	0.01	4	0.19
not_3	2	3	3	6	0.02	6	0.29
not_4	2	4	4	8	0.02	8	0.5
not_6	2	6	6	12	0.03	12	0.14
not_8	2	8	8	16	0.05	16	0.67
not_9	2	9	9	18	0.07	18	1.57
and	3	2	4	6	0.02	6	0.08
and_3	3	3	9	16	0.04	18	0.13
and_4	3	4	16	26	0.08	43	0.6
and_6	3	6	24	41	0.15	155	14.35
and_8	3	8	64	96	0.57	622	351.16
and_9	3	9	81	134	1.07	1294	1777
msign	3	4	16	34	0.08	54	0.6
fulladder	5	2	8	52	0.29	52	0.38
b10m	4	10	100	362	14.83	—	—
allen	3	13	409	498	31.16	—	—

The *fork*, *t*, *msign*, *allen*, and *fulladder* constraints represent the previously described constraints. The not_i and and_i constraints, where $i \in \{3,4,6,8,9\}$, represent the usual *not* and *and* operators for multi-valued logics. The *b10m* constraint is the multiplication of digits from 0 to 9, i.e. $b10m(X,Y,C,Z)$ stands for the constraint $X * Y = Z + 10 * C$ defined over the intervals [0..9]. The '—' symbol means that we were unable to generate the rules in less than 24 hours.

The constraints for multi-valued logics are presented in a way that shows the impact of the domain size, and of the cardinality of the constraint in the case of the same arity and a similar structure.

In spite of its exponential running time, the Equality Rules Generation algorithm is still usable. On the other hand, the Membership Rules Generation algorithm is much more costly for larger problems. Sometimes it also generates too many rules for medium size problems (such as and_9), and thus becomes unusable.

In general, it is difficult to decide which notion of local consistency should be used to solve a given CSP. In particular, Sabin & Freuder (1994) showed that in the case of CSPs consisting of binary constraints, maintaining full arc consistency during the backtracking search can be often more efficient than a more limited of constraint propagation embodied in so-called forward checking. However, empirical

results for CSPs involving non-binary are missing and it is quite conceivable that for such CSPs imposing full arc consistency during the backtracking search can be too costly. For these CSPs, a weaker form of constraint propagation, such as rule consistency, could be an alternative.

References

Abdennadher, S. & Rigotti, C. (2000) Automatic generation of propagation rules for finite domains. In: Dechter, R. (ed.), *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP'00)*, pp. 18–34. *Lecture Notes in Computer Science 1894.* Springer-Verlag.

Agrawal, R., Mannila, H., Srikant, R., Toivonen & Verkamo, A. I. (1996) Fast discovery of association rules. In: Fayyad, U. M., Piatetsky-Shapiro, G., Smyth, P. & Uthurusamy, R. (eds.), *Advances in Knowledge Discovery and Data Mining*, pp. 307–328. AAAI Press.

Allen, J. F. (1983) Maintaining knowledge about temporal intervals. *Comm. ACM*, **26**(11), 832–843.

Apt, K. R. (1998) A proof theoretic view of constraint programming. *Fundamenta informaticae*, **33**(3), 263–293. (Available via `http://arXiv.org/archive/cs/`.)

Apt, K. R. (1999a) The essence of constraint propagation. *Theoretical Computer Science*, **221**(1–2), 179–210. (Available via `http://arXiv.org/archive/cs/`.)

Apt, K. R. (1999b) The rough guide to constraint propagation. In: Jaffar, J. (ed.), *Fifth International Conference on Principles and Practice of Constraint Programming (CP'99)*, pp. 1–23. *Lecture Notes in Computer Science 1713.* Springer-Verlag. (Available via `http://arXiv.org/archive/cs/`.)

Apt, K. R. & Monfroy, E. (1999) Automatic generation of constraint propagation algorithms for small finite domains. In: Jaffar, J. (ed.), *Fifth International Conference on Principles and Practice of Constraint Programming (CP'99)*, pp. 58–72. *Lecture Notes in Computer Science.* Springer-Verlag. (Available via `http://arXiv.org/archive/cs/`.)

Apt, K. R. (2000) Some remarks on Boolean constraint propagation. In: Apt, K. R., Kakas, A. C., Monfroy, E. & Rossi, F. (eds.), *New Trends in Constraints: Lecture Notes in Artificial Intelligence 1865*, pp. 91–107. Springer-Verlag. (Available via `http://arXiv.org/archive/cs/`.)

Bessière, C. (1999) Non-binary constraints. In: Jaffar, J. (ed.), *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP'99): Lecture Notes in Computer Science 1713*, pp. 24–27. Springer-Verlag.

Bessière, C. & Régin, J. C. (1997) Arc consistency for generalized constraint networks: preliminary results. *Proceedings International Joint Conference on Artificial Intelligence (IJCAI'97)*, p. 398–404.

Borovansky, P., Kirchner, C., Kirchner, H., Moreau, P.-E. & Ringeissen, Ch. (1998) An Overview of ELAN. In: Kirchner, C. & Kirchner, H. (eds.), *Proceedings Second International Workshop on Rewriting Logic and its Applications: Electronic Notes in Theoretical Computer Science 15*, pp. 398–404. Elsevier.

By, T. (1997) *Line labelling by meta-programming.* Technical report CS-97-07, University of Sheffield, UK.

Caseau, Y. & Laburthe, F. (1996) *Introduction to the CLAIRE programming language.* Technical report, Departement Mathématiques et Informatique, Ecole Normale Supérieure, Paris, France.

Castro, C. (1998) Building constraint satisfaction problem solvers using rewrite rules and strategies. *Fundamenta informaticae*, **33**(3), 263–293.

Davis, E. (1987) Constraint propagation with interval labels. *Artif. Intell.*, **32**(3), 281–331.

Dechter, R. & van Beek, P. (1997) Local and global relational consistency. *Theoretical Computer Science*, **173**(1), 283–308.

Fleming, J. (2000) *Rainy day games.* Dell Logic Puzzles 67, 7.

Forgy, C. L. (1981) *The OPS5 user's manual.* Technical report CMU-CS-81-135, Carnegie-Mellon University.

Frühwirth, T. (1998) Theory and practice of constraint handling rules. *J. Logic Programming*, **37**(1–3), 95–138. (Special Issue on Constraint Logic Programming (P. Stuckey and K. Marriot, eds.).)

Frühwirth, T. (1995) Constraint handling rules. In: Podelski, A. (ed.), *Constraint programming: Basics and trends: Lecture Notes in Computer Science 910*, pp. 90–107. Springer-Verlag.

Kirchner, C. & Ringeissen, C. (1998) Rule-based constraint programming. *Fundamenta Informaticae*, **34**(3), 225–262.

Kleene, S. C. (1952) *Introduction to Metamathematics.* van Nostrand.

Luger, G. F. & Stubblefield, W. A. (1998) *Artificial Intelligence. 3rd ed.* Addison-Wesley.

Mackworth, A. (1977a) Consistency in networks of relations. *Artif. Intell.*, **8**(1), 99–118.

Mackworth, A. K. (1977b) On reading sketch maps. *Proceedings International Joint Conference on Artificial Intelligence (IJCAI'77)*, pp. 598–606.

Marriott, K. & Stuckey, P. (1998) *Programming with Constraints.* MIT Press.

Mohr, R. & Masini, G. (1988) Good old discrete relaxation. In: Kodratoff, Y. (ed.), *Proceedings of the 8th European Conference on Artificial Intelligence (ECAI)*, pp. 651–656. Pitman.

Montanari, U. & Rossi, F. (1991) Constraint relaxation may be perfect. *Artif. Intell.*, **48**, 143–170.

Muggleton, S. & de Raedt, L. (1994) Inductive logic programming: Theory and methods. *J. Logic Programming*, **19, 20**, 629–680.

Muth, P. (1976) A nine-valued circuit model for test generation. *IEEE Trans. Computers*, **C-25**(6), 630–636.

Pedrycz, W. & Gomide, F. (1998) *An Introduction to Fuzzy Sets.* MIT Press.

Quinlan, J. R. (1993) *C4.5: Programs for Machine Learning.* Morgan Kaufmann.

Ringeissen, C. & Monfroy, E. (2000) Generating propagation rules for finite domains: a mixed approach. In: Apt, K. R., Kakas, A. C., Monfroy, E. & Rossi, F. (eds.), *New Trends in Constraints: Lecture Notes in Artificial Intelligence 1865*, pp. 150–172. Springer-Verlag.

Sabin, D. & Freuder, E. (1994) Contradicting conventional wisdom in constraint satisfaction. In: Borning, A. (ed.), *PPCP'94: Second Workshop on Principles and Practice of Constraint Programming.*

Tsang, E. (1993) *Foundations of Constraint Satisfaction.* Academic Press.

Ullman, J. D. (1988) *Principles of Database and Knowledge-base Systems, Vol. i.* Computer Science Press.

Van Hentenryck, P., Simonis, H. & Dincbas, M. (1992) Constraint satisfaction using constraint logic programming. *Artif. Intell.*, **58**, 113–159.

Waltz, D. L. (1975) Generating semantic descriptions from drawings of scenes with shadows. In: Winston, P. H. (ed.), *The Psychology of Computer Vision*, pp. 19–91. McGraw Hill.

Winston, P. H. (1992) *Artificial Intelligence. 3rd ed.* Addison-Wesley.