Boolean and Finite Domain solvers compared using Self Referential Quizzes

Antonio J. Fernández
Lenguajes y Ciencias de la Computación,
Universidad de Málaga,
29071 Teatinos, Málaga, Spain
e-mail: afdez@lcc.uma.es
Fax-number: +34-5-2131397

Patricia M. Hill
School of Computer Studies,
University of Leeds,
Leeds, LS2 9JT, England
e-mail: hill@scs.leeds.ac.uk

Abstract

Currently there appears to be no impartial set of guidelines for choosing an appropriate constraint language for solving a specific constraint satisfaction problem. A wrong choice can cause disastrous solutions, not only relative to efficient performances, but also with respect to the code clarity of the solution, which can be important for future modifications. In this paper we make a comparative study between a significant set of constraint logic programming languages in the setting of finite domains. The chosen benchmarks consist of some unusual puzzles incorporating difficult and interesting aspects of constraint solving. By providing different solver implementations over the boolean domain, we compare different constraint languages for both expressiveness and efficiency. Furthermore, to evaluate the boolean domain, an alternative more compact finite domain representation than the boolean solution is also studied.

Keywords: Constraint Programming, Finite Domain, Choice-Point, Constraint Propagator, Labeling Strategy, Solvers.

1 Introduction

Constraint Logic Programming (CLP) keeps Logic Programming's declarativeness but provides the capacity for propagating constraints on specific domains. Among the usual constraints domains, the Finite Domain (FD) [13] constitutes an adequate framework for solving discrete constraint satisfaction problems. In this paper we make a comparative study which is an independent and objective assessment of the relative merits of CLP over FD.

Traditionally, CLP languages have provided high-level predicates (constraints) for performing specialised tasks very efficiently; their implementation is hidden and, for this reason, such solvers are called *black boxes*. The alternative *glass box* approach, allows the user to control the constraint solver at a more detailed level [14]. Here we consider two *black box* constraint languages (ECL^iPS^e [1] and Oz [12]) and two *glass box* languages (clp(FD) [5] and CHR [8]), comparing both their expressiveness and their efficiency, with particular emphasis on the expressiveness. We chose these particular languages since all provide support for FD and boolean constraints, were free[1], easy to obtain and very popular within the CLP community.

[1]Note this is the reason of the absence of commercial systems such as CHIP or ILOG Solver.

1. The first question whose answer is A is: (A) 4 (B) 3 (C) 2 (D) 1 (E) none of the above
2. The only two consecutive questions with identical answers are:
(A) 3 and 4 (B) 4 and 5 (C) 5 and 6 (D) 6 and 7 (E) 7 and 8
3. The next question with answer A is: (A) 4 (B) 5 (C) 6 (D) 7 (E) 8
4. The first even numbered question with answer B is: (A) 2 (B) 4 (C) 6 (D) 8 (E) 10
5. The only odd numbered question with answer C is: (A) 1 (B) 3 (C) 5 (D) 7 (E) 9
6. A question with answer D:
(A) comes before this one, but not after this one (B) comes after
this one, but not before this one (C) comes before and after this one
(D) does not occur at all (E) none of the above
7. The last question whose answer is E is: (A) 5 (B) 6 (C) 7 (D) 8 (E) 9
8. The number of questions whose answers are consonants is: (A) 7 (B) 6 (C) 5 (D) 4 (E) 3
9. The number of questions whose answers are vowels is: (A) 0 (B) 1 (C) 2 (D) 3 (E) 4
10.The answer to this question is: (A) A (B) B (C) C (D) D (E) E

Figure 1: The puzzle SRQ

Although a number of traditional benchmark problems have been studied with solutions implemented in different FD languages, most comparison work has been by the language implementers themselves. The Self-Referential Quiz (SRQ), which is the main benchmark used here and defined in Figure 1, is one of a new class of puzzles first described in [9] where they were expressed as satisfiability problems with solutions in Oz. The SRQs are particularly appropriate since they incorporate a number of aspects that challenge both the expressiveness and efficiency of the solvers.

It is well known that the choice of representation can have a dramatic effect on the efficiency of the solution of a problem. Therefore two different representations were used. The first uses the boolean domain and represents each option for each question directly as a boolean variable, requiring fifty such variables. The second requires the finite domain $\{1, 2, 3, 4, 5\}$ and uses an approach similar to the usual representation for the n-queens problem. This representation requires just ten variables to represent the solution.

The contributions of the paper are as follows. The main contribution is to compare the expressive power and efficiency of four popular but very different FD solvers. This can help the best choice of an appropriate language for a particular problem. The second contribution is to contrast the boolean representation with a more compact FD representation showing how the change of representation affects the results. In addition to these contributions, this study provides insights into where further research in the development of new CLP languages should be directed.

The rest of the paper is organised as follows: Section 2 explains why SRQs are suitable for our comparison and describes the original approach to solve the SRQ and a new approach, similar to the well-known FD solution to the n-queens. In Section 3, by focusing on small components of the SRQ, a study is made of the expressiveness of the different solvers. Section 4 compares the efficiency. A short conclusion and a discussion of future work end this paper.

2 Self referential quiz (SRQ): description

2.1 Why Self Referential puzzles and why these solutions?

Although there are other kinds of applications of constraint programming over FD, SRQs are particularly appropriate since they incorporate a number of aspects that challenge

$A_1 \equiv A_4 \wedge \neg A_1 \wedge \neg A_2 \wedge \neg A_3,$
$B_1 \equiv A_3 \wedge \neg A_1 \wedge \neg A_2,$
$C_1 \equiv A_2 \wedge \neg A_1,$
$D_1 \equiv A_1,$
$E_1 \equiv \neg A_1 \wedge \neg A_2 \wedge \neg A_3 \wedge \neg A_4,$

$0 \equiv (A_1 \equiv A_2) \wedge (B_1 \equiv B_2) \wedge (C_1 \equiv C_2) \wedge (D_1 \equiv D_2) \wedge (E_1 \equiv E_2),$
$0 \equiv (A_2 \equiv A_3) \wedge (B_2 \equiv B_3) \wedge (C_2 \equiv C_3) \wedge (D_2 \equiv D_3) \wedge (E_2 \equiv E_3),$
$A_2 \equiv (A_3 \equiv A_4) \wedge (B_3 \equiv B_4) \wedge (C_3 \equiv C_4) \wedge (D_3 \equiv D_4) \wedge (E_3 \equiv E_4),$
$B_2 \equiv (A_4 \equiv A_5) \wedge (B_4 \equiv B_5) \wedge (C_4 \equiv C_5) \wedge (D_4 \equiv D_5) \wedge (E_4 \equiv E_5),$
$C_2 \equiv (A_5 \equiv A_6) \wedge (B_5 \equiv B_6) \wedge (C_5 \equiv C_6) \wedge (D_5 \equiv D_6) \wedge (E_5 \equiv E_6),$
$D_2 \equiv (A_6 \equiv A_7) \wedge (B_6 \equiv B_7) \wedge (C_6 \equiv C_7) \wedge (D_6 \equiv D_7) \wedge (E_6 \equiv E_7),$
$E_2 \equiv (A_7 \equiv A_8) \wedge (B_7 \equiv B_8) \wedge (C_7 \equiv C_8) \wedge (D_7 \equiv D_8) \wedge (E_7 \equiv E_8),$
$0 \equiv (A_8 \equiv A_9) \wedge (B_8 \equiv B_9) \wedge (C_8 \equiv C_9) \wedge (D_8 \equiv D_9) \wedge (E_8 \equiv E_9),$
$0 \equiv (A_9 \equiv A_{10}) \wedge (B_9 \equiv B_{10}) \wedge (C_9 \equiv C_{10}) \wedge (D_9 \equiv D_{10}) \wedge (E_9 \equiv E_{10}),$

$A_3 \equiv A_4,$
$B_3 \equiv A_5 \wedge \neg A_4,$
$C_3 \equiv A_6 \wedge \neg A_4 \wedge \neg A_5,$
$D_3 \equiv A_7 \wedge \neg A_4 \wedge \neg A_5 \wedge \neg A_6,$
$E_3 \equiv A_8 \wedge \neg A_4 \wedge \neg A_5 \wedge \neg A_6 \wedge \neg A_7,$

$A_4 \equiv B_2,$
$B_4 \equiv B_4 \wedge \neg B_2,$
$C_4 \equiv B_6 \wedge \neg B_2 \wedge \neg B_4,$
$D_4 \equiv B_8 \wedge \neg B_2 \wedge \neg B_4 \wedge \neg B_6,$
$E_4 \equiv B_{10} \wedge \neg B_2 \wedge \neg B_4 \wedge \neg B_6$

$A_5 \equiv C_1 \wedge \neg C_3 \wedge \neg C_5 \wedge \neg C_7 \wedge \neg C_9,$
$B_5 \equiv \neg C_1 \wedge C_3 \wedge \neg C_5 \wedge \neg C_7 \wedge \neg C_9,$
$C_5 \equiv \neg C_1 \wedge \neg C_3 \wedge C_5 \wedge \neg C_7 \wedge \neg C_9,$
$D_5 \equiv \neg C_1 \wedge \neg C_3 \wedge \neg C_5 \wedge C_7 \wedge \neg C_9,$
$E_5 \equiv \neg C_1 \wedge \neg C_3 \wedge \neg C_5 \wedge \neg C_7 \wedge C_9,$

$A_6 \equiv Before^D \wedge \neg After^D$
$B_6 \equiv \neg Before^D \wedge After^D$
$C_6 \equiv Before^D \wedge After^D$
$D_6 \equiv \sum_{i \in \{1 \ldots 10\}} D_i = 0,$
$E_6 \equiv D_6$

$A_7 \equiv E_5 \wedge \neg E_6 \wedge \neg E_7 \wedge \neg E_8 \wedge \neg E_9 \wedge \neg E_{10},$
$B_7 \equiv E_6 \wedge \neg E_7 \wedge \neg E_8 \wedge \neg E_9 \wedge \neg E_{10},$
$C_7 \equiv E_7 \wedge \neg E_8 \wedge \neg E_9 \wedge \neg E_{10},$
$D_7 \equiv E_8 \wedge \neg E_9 \wedge \neg E_{10},$
$E_7 \equiv E_9 \wedge \neg E_{10},$

$A_8 \equiv \sum_{i \in \{1 \ldots 10\}} B_i + C_i + D_i = 7,$
$B_8 \equiv \sum_{i \in \{1 \ldots 10\}} B_i + C_i + D_i = 6,$
$C_8 \equiv \sum_{i \in \{1 \ldots 10\}} B_i + C_i + D_i = 5,$
$D_8 \equiv \sum_{i \in \{1 \ldots 10\}} B_i + C_i + D_i = 4,$
$E_8 \equiv \sum_{i \in \{1 \ldots 10\}} B_i + C_i + D_i = 3,$

$A_9 \equiv \sum_{i \in \{1 \ldots 10\}} A_i + E_i = 0,$
$B_9 \equiv \sum_{i \in \{1 \ldots 10\}} A_i + E_i = 1,$
$C_9 \equiv \sum_{i \in \{1 \ldots 10\}} A_i + E_i = 2,$
$D_9 \equiv \sum_{i \in \{1 \ldots 10\}} A_i + E_i = 3,$
$E_9 \equiv \sum_{i \in \{1 \ldots 10\}} A_i + E_i = 4,$

Figure 2: SRQ as a Satisfiability Problem using 50 variables

both the expressiveness and efficiency of the solvers. Our purpose is to evaluate and compare a very wide set of FD and Boolean propagators incorporated in each of the languages studied here. For that reason, although more efficient solutions can be devised[2], the two SRQ solutions shown in this paper require, in particular, a wide and significant set of different propagators over FD and Boolean variables, important for our comparative study.

2.2 The original idea

The SRQ shown in Figure 2 shows the formulation of the SRQ as satisfiability problem. Each question has five options, and each of these options is expressed as a logical formula using the connectives: conjunction, disjunction, negation and equivalence. There are 50 boolean variables l_{ij} $(i \in \{1 \ldots 10\}$ and $j \in \{A, B, C, D, E\})$ where l_{ij} has the value true if the answer to question i is j and false otherwise. Thus we call this formulation the *50 variables solution*. Note than in Figure 2 only 9 of the questions have a formulation. Question 10 is redundant and does not contribute to the solution. The additional variable $Before^D$ is defined to have the truth value of $D_1 \vee D_2 \vee D_3 \vee D_4 \vee D_5$, and $After^D$ is defined to have the truth value of $D_7 \vee D_8 \vee D_9 \vee D_{10}$.

Only one alternative may be true for each question. Thus, we also have the constraints

$$A_j + B_j + C_j + D_j + E_j = 1 \ \ (j \in \{1 \ldots 10\}) \tag{1}$$

[2]Such as that suggested by Mark Wallace (personal communication), now available in [7].

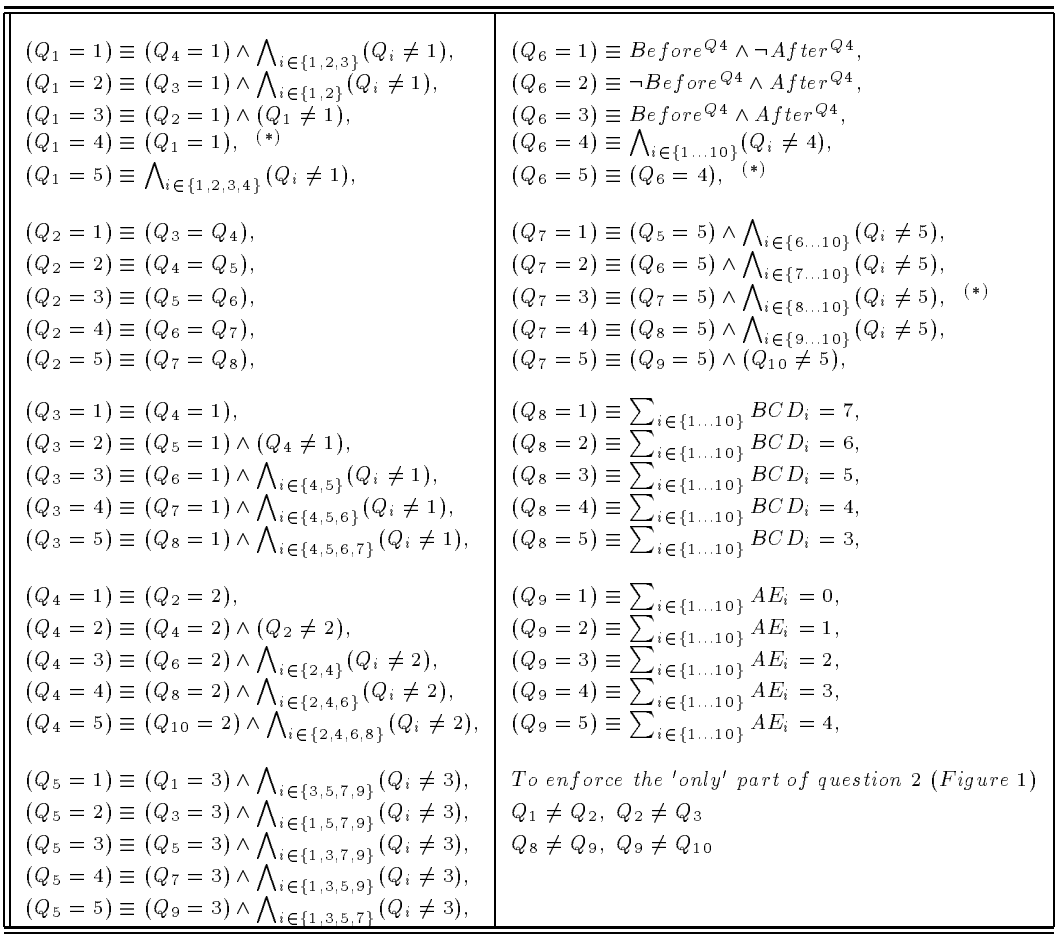

Figure 3: SRQ as a Satisfiability Problem using 10 variables

The problem consists in finding an assignment of the variables A_i, B_i, C_i, D_i and E_i ($i \in \{1 \ldots 10\}$) to truth values such that formula (1) and all the formulas in Figure 2 hold. This formulation was translated to an Oz program in [9]. In Figure 4 the table to the left shows the solution to SRQ by this approach.

2.3 An alternative approach

A more compact representation would have a single variable for each question and assign the code for the correct answer for that question to the variable (in the style of the usual representation for the n-queens problem). For that reason, a formulation for SRQ involving only 10 FD variables, which we call the *10 variables solution* and shown in Figure 3, is studied here. There is exactly one finite domain variable Q_i ($i \in \{1, \ldots, \}$) for each of the ten questions. Each Q_i takes a value in the domain 1..5 such that the answer to the $i'th$ question is in the position of Q_i in the list $[A, \ldots, E]$. The problem comprises finding a value for each of the Q_i such that all the formulas in Figure 3 hold. The additional variable BCD_i (in the formula for question 8) is 1 if $Q_i \in \{2, 3, 4\}$ and 0 otherwise and AE_i (in the formula for question 9) is 1 if $Q_i \in \{1, 5\}$ and 0 otherwise ($i \in \{1 \ldots 10\}$). The variable $Before^{Q4}$ has the truth value of $\bigvee_{i \in \{1 \ldots 5\}} (Q_i = 4)$, and $After^{Q4}$ the truth value of $\bigvee_{i \in \{7 \ldots 10\}} (Q_i = 4)$. The table to the right in Figure 4 shows the solution by this approach.

Note the constraints marked with (*) in the Figure 3 are obviously inconsistent be-

	A	B	C	D	E	Q
1	0	0	1	0	0	3
2	1	0	0	0	0	1
3	0	1	0	0	0	2
4	0	1	0	0	0	2
5	1	0	0	0	0	1
6	0	1	0	0	0	2
7	0	0	0	0	1	5
8	0	1	0	0	0	2
9	0	0	0	0	1	5
10	0	0	0	1	0	4

Figure 4: Solutions to SRQ using 50 and 10 variables

cause they constrain an FD variable to have more than one value at the same time. We observe that such constraints can lead to the removal of inconsistent values from the domain of the variables before any choices are made (meaning a priori-pruning in the search space).

The 10 variables solution uses a form of meta-constraint, that is, a constraint over constraints. This means that the constraint logical connectives are applied on constraint expressions where a constraint expression is either an arithmetic constraint (in the way of $Q_i = n$) or a combination of constraint expressions using the logical FD connectives.

3 Comparing the expressiveness of the solvers

In this section we contrast the expressiveness of the four chosen languages: clp(FD), ECL^iPS^e, Oz and CHR by implementing the 50 and 10 variables solutions to the SRQ in each of these languages. The clp(FD), ECL^iPS^e and Oz solvers are described in Subsection 3.1 and compared in Subsection 3.2. The CHR solvers are considered separately in Subsection 3.3 since the CHR language is so flexible that CHR solutions can be written in each of the styles of the other languages.

Before discussing the actual expressiveness, we briefly mention the ease with which we could learn the different CLP systems. This of course depends on many factors, including the learners background, availability of helpful documentation and personal tuition. From our perspective (used to both Prolog and functional programming languages), we found clp(FD) the easiest to master and CHR the hardest. In clp(FD) there was one main constraint to learn and this was in the form of a (declarative) Prolog predicate. We found the black-box constraints in Eclipse straightforward to use since they were built on Prolog and provided useful high-level tools needed for the problem. With Oz, there is more to learn before the language can be used effectively, but, in our case, it had the advantage that we already had an Oz implementation of the problem.

3.1 Describing the solvers

Oz. The Oz solutions are closest to the SRQ formulations shown in Figures 2 and 3. The 50-variables solver uses a *reified* constraint form to express the clauses of Figure 2. Constraints in *reified* form means that their fulfilment is reflected back into a FD variable. For example, $X = (Y + Z >: V)$ constrains X to 1 as soon as the inequation is known to be true and to 0 as soon as the inequation is known to be false. On the other hand, constraining X to 1 imposes the inequation, and constraining X to 0 imposes its negation. For instance, the clausal formula relative to the variable A_6 in Figure 2 is written in the

following way

$$
\begin{array}{l}
BeforeD = D.1 + D.2 + D.3 + D.4 + D.5 \\
AfterD = D.7 + D.8 + D.9 + D.10 \\
A.6 = BeforeD * \tilde{} AfterD
\end{array} \tag{2}
$$

The arithmetic Oz propagators $*$, $+$ and $\tilde{}$ were redefined as the boolean propagators conjunction, disjunction and negation. The 10-variables solver uses meta-constraints combined with *reified* constraints. For instance, the first logical formula relative to Q_6 in Figure 3 is expressed as follows[3]

$$
\begin{array}{l}
BeforeQ4 = ((Q1 =: 4) + (Q2 =: 4) + (Q3 =: 4) + (Q4 =: 4) + (Q5 =: 4)) \\
AfterQ4 = ((Q7 =: 4) + (Q8 =: 4) + (Q9 =: 4) + (Q10 =: 4)) \\
(Q6 =: 1) = (BeforeQ4 * \tilde{} AfterQ4)
\end{array} \tag{3}
$$

ECL^iPS^e. ECL^iPS^e does not admit a *reified* constraint form. For that reason, solvers with both 50 and 10 variables were written using meta-constraints. For instance, the 50-variables solver codes the formula relative to the variable A_6 in Figure 2 as follows

$$
\begin{array}{l}
(BeforeD\#= 1)\# \Leftrightarrow (D1\# = 1)\#\backslash/(D2\# = 1)\#\backslash/(D3\# = 1)\#\backslash/(D4\# = 1)\#\backslash/(D5\# = 1), \\
(AfterD\# = 1)\# \Leftrightarrow (D7\# = 1)\#\backslash/(D8\# = 1)\#\backslash/(D9\# = 1)\#\backslash/(D10\# = 1), \\
(A6\# = 1)\# \Leftrightarrow (BeforeD\# = 1)\#/\backslash(AfterD\# = 0)
\end{array}
$$

$\#/\backslash$ and $\#\backslash/$ are conjunction and disjunction propagators respectively, and $\# \Leftrightarrow$ is the equivalence propagator between constraint expressions whereas $\# =$ is the arithmetic equivalence propagator between rational terms[4].

In a similar way, the 10-variables solver writes the formula relative to Q_6 as follows

$$
\begin{array}{l}
(BeforeQ4\# = 1)\# \Leftrightarrow (Q1\# = 4)\#\backslash/(Q2\# = 4)\#\backslash/(Q3\# = 4)\#\backslash/(Q4\# = 4)\#\backslash/(Q5\# = 4), \\
(AfterQ4\# = 1)\# \Leftrightarrow (Q7\# = 4)\#\backslash/(Q8\# = 4)\#\backslash/(Q9\# = 4)\#\backslash/(Q10\# = 4), \\
(Q6\# = 1)\# \Leftrightarrow (BeforeQ4\# = 1)\#/\backslash(AfterQ4\# = 0)
\end{array}
$$

clp(FD). The 50-variables solution shows little resemblance to the clausal formulation of SRQ in Figure 2 because the code is closer to traditional Prolog style. For instance, the logical formula relative to the variable A_6 in Figure 2 is written as follows:

$$
\begin{array}{l}
or(D1, D2, DD12), or(D3, D4, DD34), or(DD12, DD34, DD1234), \\
or(DD1234, D5, BeforeD), or(D7, D8, DD78), or(D9, D10, DD910), \\
or(DD78, DD910, AfterD), not(AfterD, AfterDn), and(BeforeD, AfterDn, A6)
\end{array}
$$

which is far from the clarity of the Oz solution. Moreover, implementing a 10-variables solution in clp(FD) highlights certain problem with the lack of expressiveness. This limitation appears when we try to express meta-constraints. For example the following constraint expresses that the variable $BeforeQ4$ in Figure 3 has the truth value of $\bigvee_{i \in \{1 \ldots 5\}} (Q_i = 4)$

[3]Note that =: is the equivalence propagator.

[4]**A rational term** is a term constructed from integers and integer domain variables using the arithmetic operations $+, -, *$ and $/$.

$$(BeforeQ4 = 1) \Leftrightarrow (Q1 = 4) \vee (Q2 = 4) \vee (Q3 = 4) \vee (Q4 = 4) \vee (Q5 = 4) \quad (4)$$

Note that the problem consists of detecting when a constraint $(BeforeQ4 = 1)$ is now true. In [3] is shown how to express directly that kind of constraints in the FD with $X\ in\ r$ constraint by the entailment of FD constraints, but this is not implemented. However, this can be done in clp(FD) by means of *user functions* written in C. These functions accept ranges or terms as argument. For instance for writing the constraint (4) we use calls to the predicate $'x = a \Leftrightarrow b'/3$. The semantics of a call $'x = a \Leftrightarrow b'(X, A, B)$ is: $X = A\ iff\ B$ is true (i.e. $B = 1$). This predicate is defined in the same way as in the magic square's solution given in [5]:

$$'x = a \Leftrightarrow b'(X, A, B)\ :-\ B\ in\ 'X\_To\_B'(dom(X), A),\ X\ in\ 'B\_To\_X'(val(B), A).$$

where $'X\_To\_B'$[5] and $'B\_To\_X'$[6] are *user functions* coded in C. Then, the constraint (4) is expressed as follows:

$$'x = a \Leftrightarrow b'(BeforeQ4, 1, B1),\ 'x = a \Leftrightarrow b'(Q1, 4, B2),\ 'x = a \Leftrightarrow b'(Q2, 4, B3),$$
$$'x = a \Leftrightarrow b'(Q3, 4, B4),\ 'x = a \Leftrightarrow b'(Q4, 4, B5),\ 'x = a \Leftrightarrow b'(Q5, 4, B6),$$
$$or(B2, B3, B23),\ or(B4, B5, B45),\ or(B23, B45, B2345),\ or(B2345, B6, B1)$$

The inclusion of C code means that the program is not fully declarative. Moreover, just for this one constraint, nine additional boolean variables (B1, B2, B3, B4, B5, B6, B23, B45, B2345) are necessary, complicating the code. Since the SRQ puzzle contains 50 options, expressed in a similar syntax to the constraint (4), the total number of extra FD variables needed for the solution is quite high. In larger puzzles the number of variables might be unmanageable. Because of this limitation, we did not implement a 10-variables solver in clp(FD).

3.2 The solvers compared

The expressiveness is analysed relative to the similarity of the code of each solver to the SRQ clausal formulations shown in Figures 2 and 3. The conclusions are the following:

Oz. Oz shows a higher expressiveness than clp(FD) and ECL^iPS^e because it

- Uses a *reified* constraint form and admits infix[7] propagators with concatenation[8].
- Propagators can operate directly over boolean variables as in (2) or over constraint expressions as in (3).

ECL^iPS^e. This also has a high expressiveness. The propagators can be written in an infix form and concatenated as in Oz. But there are two differences with Oz. First, some propagators are limited to operate on constraint expressions over FD variables and not directly over these. For instance, to express that the negation of a boolean variable X must be true, it is not possible to write simply ~X (as Oz does), but $(X\# = 0)$ or $(X\backslash\# = 1)$. Secondly, the code required some additional variables to write the boolean

[5] This function returns 1 if $X = A$, O if $X \neq A$ and 0..1 otherwise.

[6] This function is triggered as soon as B is instantiated and yields A if $B = 1$ or else the range $0..\infty\backslash A$.

[7] To the style of $arg_i\ R\ arg_j$, where R is a constraint propagator, and arg_i and arg_j its arguments.

[8] In the way $arg_0\ R_1\ arg_1\ R_2 \ldots R_{n-1}\ arg_{n-1}\ R_n\ arg_n$, where R_i is a propagator $(i \in \{1 \ldots n\})$.

domain solution (these are required for the summation of boolean variables).

clp(FD). The 50-variables solution shows little resemblance to the clausal formulation of the SRQ shown in Figure 2. The reasons are the following:

- The code includes a large number of extra FD variables.
- Most of the constraint propagators have a relational prefix form[9] close to the writing style of traditional Prolog. Thus, it is not possible to employ the concatenation shown in ECL^iPS^e and Oz which clarifies the code and would save the addition of extra variables.
- Some constraints cannot be expressed in a direct way using built-in predicates. For instance, when we tried to express $D_6 = \sum_{i \in \{1...10\}} D_i = 0$, (see Figure 2) the same problem mentioned in Subsection 3.1 was found. This constraint is expressed in form of meta-constraints $(\sum_{i \in \{1...10\}} D_i = 0) \Leftrightarrow (D6 = 1)$. To solve this, it was necessary to use the predicate $'x = a \Leftrightarrow b'/3$ (see Subsection 3.1) as follows:

$$SmD = D1 + D2 + D3 + D4 + D5 + D6 + D7 + D8 + D9 + D10,$$
$$'x = a \Leftrightarrow b'(SmD, 0, D6),$$

$+$ is the summation propagator for FD variables. Questions 8 and 9 were written analogously. Note that it is not possible use the built-in equivalence propagator *equiv/3* [5] over the variable SmD because its domain is the range 0..10 (and $equiv/3$ operates only in the boolean domain).

3.3 CHR: a special mention

Expressively speaking, CHR deserves a special consideration due to its flexibility for writing solvers. Its particular glass-box approach allows one to express the same solvers in different styles. To demonstrate this flexibility, the clp(FD) solution was used as a model for a CHR solver involving 50 boolean variables. The different clp(FD) boolean propagators were simulated by the CHR constraints. For example, the clp(FD) symbolic boolean constraint *only_one(+L)* [6] (used to implement the formula (1)) which is true if exactly one boolean variable of the list L is equal to 1, was coded in CHR as follows:

$$only\_one(L) \Leftrightarrow atleast(1, L, 1), atmost(1, L, 1).$$

where $\Leftrightarrow$ defines the simplification over user-defined constraints in CHR. The call *atmost(1,L,1)* is true if at most one element of the list L is equal to 1 and the *atleast/3* predicate was defined in a similar way.

Meta-constraints and *reified* constraints are no problem in CHR. We also implemented the CHR solver involving 10 FD variables using a style similar to the solution in Oz. We define an equivalence propagator between constraint expressions by means of the definition of a *solve/2* predicate which receives in its first argument a logical restriction over variable constraints (that is a meta-constraint) or FD variables and returns in its second argument the result of it. To impose a constraint it is enough to impose a true value in the last argument and write the constraint as the first argument. For instance, this CHR solver expresses in a similar style to Oz the formula relative to Q_6 in Figure 3 as follows:

[9] In the way of *R(X,Y,Result)*, where *R* is a propagator and *X*, *Y* and *Result* are FD boolean variables.

$solve(BeforeQ4 \Leftrightarrow (Q1 = 4) + (Q2 = 4) + (Q3 = 4) + (Q4 = 4) + (Q5 = 4), 1),$
$solve(AfterQ4 \Leftrightarrow (Q7 = 4) + (Q8 = 4) + (Q9 = 4) + (Q10 = 4), 1),$
$solve((Q6 = 1) \Leftrightarrow (BeforeQ4 * \tilde{} AfterQ4), 1).$

4 Experimental results

4.1 Implementations over boolean domain

Here, we present the performances of the set of solvers for the SRQ implemented under the approach shown in Subsection 2.2. SRQ solutions have been implemented in ECL^iPS^e[10], CHR[11] and clp(FD)[12]. We have modified the original Oz solution in its labeling strategy[13] because we wanted to do the efficiency comparisons under the same conditions of labeling. Two of the most popular labeling strategies have been chosen. Every solver involves 50 boolean variables in the same way that the original solution in Oz, and Table 1 summarises the results. Note Oz is taken as reference because it provides the original solution. All the results have been calculated using the same 4/50 SPARCstation IPX (40 MHz).

language	*labeling strategy*	*time first*	*time all*	*speedup*
clp(FD)	indomain	90	110	4.62
	ff	80	110	5.00
ECL^iPS^e	indomain	850	1000	$\downarrow$ 2.04
	ff	933	1083	$\downarrow$ 2.33
CHR	chr	6150		(in) $\downarrow$ 14.78
				(ff) $\downarrow$ 15.37
Oz	indomain	416	434	1.00
	ff	400	416	1.00

Table 1: Performance results of the 50-variables solvers for the SRQ

The meaning of the columns is as follows. The first column gives the name of the constraint language used in the implementation. The next column gives the labeling strategy used. The word *indomain* means that a consistent value to a FD variable has been assigned from the minimum of its domain (through backtracking, all possibles values can be enumerated). The word *ff* indicates the use of the *first fail principle* heuristic. The word *chr* means the built-in labeling feature for constraint handling rules (CHR) has been used. The third column gives the running time to find the first (and unique) answer, measured in milliseconds. The next column shows the time to explore the whole search space. The last column gives the average speed-up in relation to the original solution in Oz. The symbol $\downarrow$ indicates that the following number is the average slow-down in relation to the Oz solution. (in) and (ff) in the rows of the CHR language means the comparison is done relative to the *indomain* and *first fail* labeling strategy respectively.

The results show clp(FD) solver is clearly the fastest, while Oz is more than twice as fast as ECL^iPS^e. The CHR solver presents the weakest results. This is to be expected since the CHR used here is implemented on the top of the ECL^iPS^e system. Because the

[10] version ECL^iPS^e 3.5.

[11] Available as library of ECL^iPS^e.

[12] version clp(FD) 2.21.

[13] When no more propagation was possible, the variable on which more propagators depend was chosen, and then its maximal value was tried first. In this case, the most suitable strategy was used and the general condition could propagate much better.

slow-down for CHR is so high we only provide the result for the *first solution* search. Note that, although we have tried to develop optimal solutions to the problem for each solver, the authors are not experts in any of the constraint languages tested. It is therefore possible that some of the differences in performance are due to an inappropriate use of the constraint systems rather than the underlying implementations themselves.

language	*labeling strategy*	*time first*	*time all*	*speedup*	*speedup prec*
ECL^iPS^e	indomain	566	900	↓ 1.36	1.50
	ff	583	933	↓ 1.45	1.60
CHR	chr	4400		(in) ↓ 10.57	1.39
				(ff) ↓ 11.00	1.39
Oz	indomain	318	418	1.30	1.30
	ff	317	400	1.26	1.26

Table 2: Comparable results of the 10-variables solution for the SRQ

4.2 Solvers under the 10 FD variables approach

Using the 10 variables formulation in Subsection 2.3, we implemented solutions to the SRQ in ECL^iPS^e, CHR, and Oz[14]. Note we have not implemented a clp(FD) under this approach due to the limitation explained in Section 3.1. Table 2 shows the performance results of these solutions. The meanings for the columns are the same as in Table 1, where again the column *speedup* shows the speed-up factor relative to the original solution in Oz with 50 boolean variables, and the additional column *speedup prec* indicates the speed-up factor with respect to the 50-variables solver implemented in the same language (results in Table 1). This gives an idea about the gain obtained under the 10-variables approach with respect to the 50-variables solver. All solvers implemented under the 10-FD variables approach improve the performances of the solvers implemented in the style of the original Oz solution involving 50-FD boolean variables. Therefore, the efficiency of the solving has been improved.

4.3 Comparing the constraint propagation

The hardness of this kind of puzzle seems to be in the number of choice-points it is necessary to do for finding a solution. For that reason, we compare the different Oz solvers under the two different approaches (50 and 10 variables) giving them to a particular inference machine (which performs the search) called Explorer tool [11] which allows to visualise the search tree and the choice nodes (circles), failure nodes (squares) and solution nodes (diamonds). The left tree in Figure 5 shows the search tree of the original Oz program for *first solution* search in the SRQ solving using the *first fail* labeling strategy. It contains 27 nodes (13 choice nodes, 13 failure ones and 1 solution one). The right tree shows, in the same conditions, the 10-variables solution which contains 9 nodes with only 5 choice nodes. This is a significant reduction by 66% in the number of nodes and 62% in the number of choices nodes. The amount of constraint propagation depends on the number of choices it is necessary to do for finding a solution. Thus, this reduction means the solution involving 10 FD variables leads to an increase in the constraint propagation.

[14]Version Oz 2.

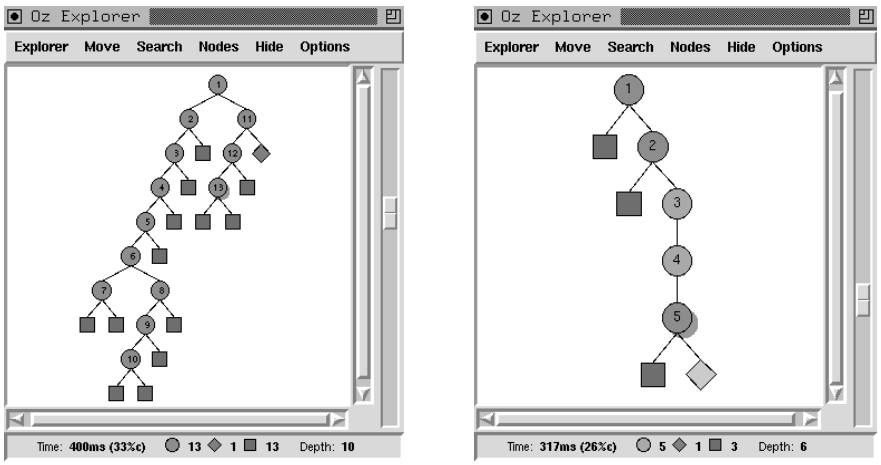

Figure 5: The Oz Explorer on SRQ solving using *first fail* labeling for *first solution* search

5 Conclusions and further work

In this paper we have provided some guidelines to make a correct election of the constraint language to use in the solving of constraint satisfaction problems. Four different languages under two different approaches (*black-box*, *glass-box*) has been compared by implementing solvers for a new puzzle that combines features of the magic square's puzzle with the n-queens' problem. From the efficiency point of view, clp(FD) is the best language for solving discrete problems involving boolean variables whereas, for the non-boolean finite domain, Oz gives a good performance. From the expressiveness point of view, we have shown the flexibility of CHR language for writing solvers. The limitations of the current version of clp(FD) for writing constraints involving implications of finite domain constraints has been highlighted[15]. Thus to summarise, for maximum efficiency use clp(FD), for maximum expressiveness use CHR, while for an equilibrium between expressiveness and efficiency, the Oz language appears best.

The SRQ and the formulations shown in Figures 2 and 3 used in this study combine a number of interesting features. Furthermore, the codes used for the solutions are available over Internet in [7]. Thus these puzzles can provide a useful benchmark for evaluating new implementations of existing languages as well as a basis for studying new language extensions.

Formal studies such as that made here could also be done to compare various special characteristics of constraint languages used in real life applications such as feature constraints, coroutining, delayed constraints, multiple head atoms, pseudo-boolean constraints, linear constraints and cumulative constraints. These studies are currently being extended to include clp(B) [4], clp(PB) [2], and clp(B/FD) [5] languages and commercial systems such as SICStus, IF/Prolog, CHIP and Ilog Solver. Solutions will be available in [7] for each of the languages.

It is clear that a language with the expressiveness of CHR but with an efficiency comparable with clp(FD) is desirable. Thus future research could consider how these glass-box approaches to constraints could be merged. Alternatively, designing an efficient implementation of a CHR solver would make the CHR approach more competitive.

[15] A possible solution could be found in the CCP (Concurrent Constraint Programming) framework [10] which allows for a new operation called Ask and that is really an if-then where the test is a constraint to be checked and the then part is any predicate (including the tell of a constraint).

Acknowledgments

We gratefully acknowledge Daniel Diaz, Mark Wallace and Gyuri Lajos for their helpful remarks. We also thank the anonymous referees for their careful reading and helpful comments.

References

[1] AGGOUN A., CHAN D., DUFRESNE P., FALVEY E., GRANT H., HEROLD A., MACARTNEY G., MEIER M., MILLER D., MUDAMBI S., PEREZ B., Van ROSSUM E., SCHIMPF J., PERIKLIS, TSAHAGEAS A. and de VILLENEUVE D.H., *ECL^{i}PS^e* 3.5, User Manual, ECRC, Munich, 1995.

[2] BARTH P., and BOCKMAYR A.,Modelling 0-1 Problems in CLP(PB). In *Proc. of the 2nd International Conference on the Practical Application of Constraint Technology*, London, 1996.

[3] CARLSON B., CARLSSON M. and DIAZ D., Entailment of Finite Domain Constraints. In *Proc. of the 11th International Conf. on Logic Programming*, Santa Margherita, 1994.

[4] CODOGNET P. and DIAZ D., clp(B): Combining Simplicity and Efficiency in Boolean Constraint Solving. In Proc. of the *6th International Symposium of Programming Language Implementation and Logic Programming (PLILP'94)*, pps: 244-260, Madrid, Spain, 1994.

[5] CODOGNET P. and DIAZ D., Compiling Constraints in *clp(FD)*. In *The Journal of Logic Programming*, 27:1-199 ,1996.

[6] DIAZ D., clp(FD) 2.21 User's Manual, 1994.

[7] FERNÁNDEZ A., http://www.lcc.uma.es/personal/fernandez_l/srq/index.html, *SRQ Solutions*, 1997.

[8] FRÜHWIRTH T., Constraint handling rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, pp:90-107, LNCS 910, May 1994.

[9] HENZ M., Don't Be Puzzled!. In *Workshop on Constraint Programming* in conjunction with the *Second International Conference on Principles and Practice of Constraint Programming (CP96)*, Cambridge, Massachussets (USA). August 1996.

[10] SARASWAT V.A., *Concurrent Constraint Programming*. The MIT Press, 1993.

[11] SCHULTE C., Solver–a search debugger for Oz. In *WOz'95, International Workshop on Oz Programming*, Martigny, Switzerland, November, 1995.

[12] SMOLKA G. and TREINEN R., editors. *DFKI Oz Documentation Series*, German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany, 1995.

[13] Van HENTENRYCK P., *Constraint Satisfaction in Logic Programming*. The MIT Press, 1989.

[14] Van HENTENRYCK P., SARASWAT V.A. and DEVILLE Y. Design, Implementation and Evaluation of the Constraint Language cc(FD). In A. Podelski, editor, *Constraint Programming: Basics and Trends*, pp:293-316, LNCS 910, May 1994.