Constraints, 5, 275–301 (2000)
© 2000 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

A Comparative Study of Eight Constraint Programming Languages Over the Boolean and Finite Domains

ANTONIO J. FERNÁNDEZ afdez@lcc.uma.es
Departamento de Lenguajes y Ciencias de la Computación, E.T.S.I.I., 29071 Teatinos, Málaga, Spain

PATRICIA M. HILL hill@scs.leeds.ac.uk
School of Computer Studies,University of Leeds, Leeds, LS2 9JT, England

Abstract. This paper compares the efficiency of a number of Constraint Logic Programming (CLP) systems in the setting of finite domains as well as a specific aspect of their expressiveness (that concerning reification and meta-constraints). There are two key reasons for adopting CLP technology for solving a problem. The first is its expressiveness enabling a declarative solution with readable code which is vital for maintenance and the second is the provision of an efficient implementation for the computationally expensive procedures. However, CLP systems differ significantly both in how solutions may be expressed and the efficiency of their execution and it is important that both these factors are taken into account when choosing the best CLP system for a particular application. This paper aids this choice by illustrating differences between the systems, indicating their particular strengths and weaknesses.

Keywords: constraint programming, constraint propagator, domain, labeling strategy, solvers

1. Introduction

Evidence of the success of the Constraint Logic Programming (CLP) paradigm ([10], [18], [20]) can be found in the increasing number of CLP systems now being used for many real-life applications [31]. There are two main reasons for this success: first, CLP extends the Logic Programming paradigm enabling more declarative and readable solutions and, secondly, it supports the propagation of constraints for specific domains, providing an efficient implementation for the computationally expensive procedures. However, CLP systems differ significantly both in how solutions may be expressed and the efficiency of their execution. It is important that both these factors are taken into account when choosing the best CLP system for a particular application. In fact, a wrong choice for an application may be disastrous, not only relative to its efficient performance, but also with respect to the code clarity of the solution, important for future modifications. In spite of this, there appears to be no impartial set of guidelines for choosing an appropriate constraint system for solving a specific constraint satisfaction problem.

Of the domains for CLP, the Finite Domain (FD) [29] is one of the best studied since it is a suitable framework for solving discrete constraint satisfaction problems. FD is particularly useful for modeling problems such as scheduling, planning, packing, timetabling and as a consequence most of the CLP systems provide substantial FD libraries. In this paper, only the FD and the Boolean domain (which is sometimes regarded as an instance of the FD) are considered.

The main contribution of this paper is to provide an impartial comparison of a number of FD solvers for CLP. We contrast eight constraint systems; ECL^{i}PS^e [1], Oz [27], Ilog SOLVER [17], clp(FD) [7], CHR [13], SICStus [25], IF/Prolog [16] and B-Prolog [32]. We chose these particular systems since they cover the main kinds of FD solvers and are all very popular within the CLP community. Because of our limited resources, CHIP [28] and Prolog IV [22] are not included in this comparison. Although the constraint systems have been tested on the solving of a number of traditional benchmarks, most comparative work has been done by the language implementers themselves ([3], [6], [7], [9], [21], [23], [26]). Furthermore, these benchmarks have been used in the development as well as the assessment of the languages so that such tests are biased. Neither of the authors of this paper is a designer of any of the CLP languages studied here so that the comparison is more impartial than previous ones. It is intended that the experience reported here will aid others in choosing an appropriate constraint language for solving their specific constraint satisfaction problem.

We have chosen a particular kind of logical puzzle, called Self-Referential Quiz (SRQ) as the original benchmark for the comparison. This is one of a new class of puzzles first described in [15] for demonstrating the meta-reasoning capabilities of the OZ FD system. Because of the self referential nature of the problem, it is particularly suitable for examining the ease with which the different languages can be used for applications requiring meta-reasoning. Thus, here we use the SRQ to illustrate how each of the languages support reification in the Boolean domain and meta-constraints in the FD. Also, we use the SRQ for the efficiency comparisons since this is a non-standard benchmark not used in the development of any of the systems (apart from Oz). Hence these particular comparisons are essentially fair and unaffected by any bias the implementation may have to perform well on the standard benchmarks.

As is discussed in Subsection 5.1, constraint solving can be viewed as a combination of two processes, constraint propagation and labeling. Since constraint propagation is the main reason for choosing the CLP technology, we concentrate on comparing the constraint propagation methods. However, to ensure fairness in the efficiency testing, we consider two distinct labelings, *naive* and *first fail*.

It is well known that a single benchmark is not adequate for evaluating a programming system. Moreover, SRQs can be solved quickly and, indeed, are not easily scalable. Thus, to make the comparison fair and thus more objective, we extended our study to other problems (two of which are scalable) comparing mainly the efficiency of the solutions. We chose some well-known problems and used, wherever available, solutions provided with the systems or, if not, directly from the implementers.

The rest of the paper is organised as follows: Section 2 describes the essential features of the FD constraint systems for the eight languages considered in this paper. In Section 3, the SRQ puzzle is defined and two different solutions to it are described. Section 4 shows, in a tutorial way, how each language can be used to express reified constraints and meta-constraints. Extracts from the SRQ formulations in each of the languages are used to highlight the main differences between them. In Section 5, a comprehensive efficiency comparison is reported and the results are discussed. The paper concludes with a summary and an outline of possible future work.

2. The Constraint Systems Tested

In this section, we describe the constraint systems considered in the paper. These systems are classified here as *glass box* or *black box*. So, first we explain what we mean by these classifications. Within each classification, we then describe the constraint systems compared. We conclude the section with a short note discussing the ease with which the languages could be learned.

2.1. Glass Box vs Black Box

The distinction between a *glass* or a *black* box language is not always clear. By *glass box* languages [30] we mean those which provide very simple and primitive constraints in which the propagation scheme can be formally specified. These constraints can then be used to construct specialised high level constraints suitable for the application. Alternatively, *black box* languages are those which provide a wide range of high level constraints whose implementation is hidden from the user. These constraints perform specific tasks very efficiently. In these languages, it is hard for a user to add new constraints since such constraints have to be defined at a low level requiring a detailed knowledge of the implementation.

2.2. Glass Box Languages

There are two different kinds of *glass box* languages. These differ in the way that constraint propagation may be defined: either using a single form of relational construct called an *indexical* [8] or by means of special *Constraint Handling Rules* (CHRs) [13].

2.2.1. Indexical Languages

The three languages examined here that support indexicals are clp(FD) 2.21, SICStus 3#5 and IF/Prolog 5.0.

An *indexical* is a reactive functional rule of the form *X in R* where *X* is a domain variable. *R* is a set-valued range expression of the form $t_1 \dots t_2$ in which terms t_1 and t_2 denote singleton ranges, parameters, integers, combinations of terms using arithmetical operators or indexical ranges. The allowed form of an indexical range depends on the language but is normally one of the following:

- min(Y) which represents the minimal value of the domain variable Y.
- max(Y) which represents the maximal value of the domain variable Y.
- val(Y) which represents the value of Y as soon as it is ground.
- dom(Y) which represents the current domain of Y.

The indexical constraint can be seen as an abstract machine for propagation-based constraint solving [8]. It is possible to directly encode most of the higher level FD constraints

with this one basic constraint. The feasibility of efficiently integrating the indexical approach with a Prolog based on the WAM has been demonstrated by an implementation by Diaz [5].

2.2.2. *A Language with Constraint Handling Rules*

For a CHR language, we use the library that is built on top of ECL^iPS^e 3.5.2, amalgamating the CHRs with the underlying language. A CHR can define *simplification* and *propagation* over user-defined constraints. A *simplification* rule replaces constraints by simpler constraints while preserving logical equivalence. For example:

$$X > Y, Y > X \Leftrightarrow \mathit{false}.$$

A *propagation* rule adds new constraints which are logically redundant but may cause further simplifications. For example:

$$X > Y, Y > Z \Rightarrow X > Z.$$

Repeatedly applying CHRs incrementally simplifies and, possibly, solves the user-defined constraints.

Note that the CHRs were originally intended as a language for constraint simplification but have since been shown to be useful for building specialised constraint solvers for particular applications and domains.

2.3. *Black Box Languages*

The four *black box* languages examined here are described below.

2.3.1. *Oz 2.0*

Oz is a new language combining functions with relations so that it has the potential for extra expressiveness in the constraint solver.

Oz provides algorithms to decide the satisfiability and implications for basic constraints which take the form $x = n$, $x = y$ or $x{:}\,:\,D$ where x and y are variables, n is an nonnegative integer and D is a finite domain. The basic constraints reside in the constraint store. Non-basic constraints, such as $x + y = z$, are not contained in the store but are imposed by *propagators* [21]. An Oz *propagator* is a computational agent which is posted on the variables occurring in the corresponding domain. It reads the constraint store and tries to narrow the domains posted there by amplifying the store with basic constraints.

For example, suppose there is a constraint store containing the domain variables X, Y with the domain $\{1, \ldots, 10\}$. The propagator for $X + Y = 5$ narrows the domain for both X and Y to $\{1, \ldots, 4\}$. The propagator $X + Y = 5$ is said to constrain the variables X

and Y. Adding the constraint $Y = 1$ narrows the domain of Y to 1 and the domain of X to 4.

Propagators are provided by both glass and black box languages. However, in glass box languages, the indexicals and CHRs are the basic components and the propagators are constructed from them. On the other hand, in black box languages, there is no means of specifying directly the constraint propagation so that propagators such as $+$ or $-$ are themselves primitives and are determined solely by their operational semantics.

2.3.2. *ECL^{i}PS^e 3.5.2*

ECL^{i}PS^e includes the traditional finite domain constraints that are now being incorporated in many logic programming systems. It also supports writing further extensions such as new user-defined constraints or complete new constraint solvers such as CHR. These extensions are based on a mechanism of suspension and waking of goals provided by ECL^{i}PS^e. To make such an extension, the user needs a good knowledge of the underlying system and this is the reason why ECL^{i}PS^e is not catalogued as a *glass box* language.

Note that, in addition to integers, the FD library allows for atomic (e.g., atoms, strings, floats) and ground compound elements (e.g., *f(a,b)*).

2.3.3. *Ilog SOLVER 3.1*

Ilog SOLVER is a C++ library for constraint programming so that the underlying data and control structures must be defined in C++. Thus it is not strictly a CLP system. However, as it uses the CLP approach and is a popular commercial system for solving constraint satisfaction problems, we include it with the systems being studied.

In Ilog SOLVER, a constraint is either an object or a Boolean expression with values false (**IlcFalse**) or true (**IlcTrue**). The actual value depends on the satisfiability of the constraint: if the constraint cannot be violated, then the expression is bound to **IlcTrue** and if the expression cannot be satisfied then it is bound to **IlcFalse**. These expressions can themselves be constrained or combined with logical operators *or*, *and* and *not* to create more complex constraints.

When a constraint is posted (by means of the function **IlcPost**), the constraint is used immediately to reduce the domains of the constrained variables that it involves. Backtracking is provided by combining non-deterministic elements.

2.3.4. *B-Prolog 2.1*

As with ECL^{i}PS^e, this system provides a set of traditional finite domain predicates such as arithmetic or Boolean constraints and a set of primitives to process the domain variables. Note that this set of built-in constraint predicates is smaller than that provided by ECL^{i}PS^e.

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 SRQ puzzle.

2.4. Ease of Learning

We now discuss the ease with which we could learn the different CLP languages based on our experience of writing solutions to the SRQ problem described in Section 3. The ease which a new language may be learned 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), SICStus and IF/Prolog the simplest to master since they are based on one main constraint and this was in the form of a (declarative) Prolog predicate. We found the *black box* constraints in ECL^{i}PS^e and B-Prolog straightforward to use since they were built on Prolog and provided useful high level tools needed for the problem. Oz had an unfair advantage in that we already had an Oz implementation of the SRQ problem. However, compared to clp(FD), SICStus and IF/Prolog, we found that more needed to be learnt before the language could be used effectively. Ilog SOLVER requires a working knowledge of C++ but, since it does not impose any syntactic extension of C++, no new language syntax needs to be learnt. For CHR, not only did we have to learn a new language syntax but also we had to understand the novel constraint propagation mechanism defined by this syntax so that this system took longest for us to master.

3. The Self Referential Quiz and Two Solutions

In this section, we describe two alternative formulations for the SRQ puzzle defined in Figure 1. One formulation uses a Boolean representation with fifty Boolean variables and the other uses a representation requiring ten variables that can range over the numbers from one to five. These two formulations are useful, not only to demonstrate the differences in performance between the Boolean and FD solutions for the different systems but also illustrate two aspects of the meta-reasoning capabilities of the languages: reification and meta-constraints. Note that this is the reason that we do not consider other, possibly more

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

Q
3
1
2
2
1
2
5
2
5
4

Figure 2. Solutions to SRQ using 50 and 10 variables.

efficient, solutions to the problem[1]. These two aspects are discussed in more detail in the next subsections.

3.1. A Representation Using Booleans

This representation defines the SRQ as a 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 representation the *50 variables formulation*. This formulation was translated to an Oz program in [15]. The left-hand table of Figure 2 shows the solution to SRQ using this representation.

3.2. A More Compact Representation Using Meta-Constraints

A more compact representation for the SRQ puzzle in Figure 1 has a single variable for each question and assigns the value for the correct answer to that question to the variable (in the style of the usual representation for the N queens problem). For that reason, a representation for SRQ involving only 10 FD variables, which we call the *10 variables formulation* is studied here. There is exactly one finite domain variable Q_i ($i \in \{1 \ldots 10\}$) for each of the ten questions. Each Q_i takes a value in the domain $1 \ldots 5$ such that the answer to the $i'th$ question is in the position Q_i in the list $[A, \ldots, E]$. The right-hand table of Figure 2 shows the solution using this representation.

See [11] for more information about the two representations described above.

4. Reification and Meta-Constraints

In this section we explain how reified constraints and meta-constraints can be coded in each of the eight languages: clp(FD), ECL^{i}PS^e, Oz, SICStus, IF/Prolog, Ilog SOLVER, B-Prolog and CHR. The first seven languages are analysed in Subsection 4.3. The CHR language is considered separately in Subsection 4.4 since the CHR language is so flexible

that CHR code can be written in each of the styles of the other languages. We use the SRQ puzzle to illustrate this.

It is important to note that we only consider one aspect of the expressiveness (that concerned with reification and meta-constraints). More comprehensive discussions concerning expressiveness should also consider, among others, aspects such as (a) the ability to express search strategies, (b) the ease of development and integration of constraint satisfaction techniques and (c) the ease of reusing previous work through the development and exploitation of additional libraries.

4.1. Reified Constraints

Reified constraints reflect the validity of a constraint into a Boolean variable. Constraints in *reified* form allow their fulfillment to be reflected back into an 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. The logical formula associated to option A in question 6 in the *50 variables formulation* for SRQ is expressed, using reified constraints, as follows:

$$A_6 \equiv BfD \wedge \neg AfD \tag{1}$$

where the variable BfD is defined to have the truth value of $D_1 \vee D_2 \vee D_3 \vee D_4 \vee D_5$, and AfD is defined to have the truth value of $D_7 \vee D_8 \vee D_9 \vee D_{10}$.

4.2. Meta-Constraints

Meta-constraints provide a means of expressing constraints over constraints. For these meta-constraints the logical connectives have to be applied directly to a constraint such as $Q = n$ or to an expression using these logical connectives. For example, the logical formula associated to option A in question 6 in the *10 variables formulation* for SRQ is expressed, using meta-constraints, as follows:

$$(Q_6 = 1) \equiv BfQ \wedge \neg AfQ, \tag{2}$$

where the variable BfQ has the truth value of $\bigvee_{i\in\{1\ldots5\}}(Q_i = 4)$, and AfQ the truth value of $\bigvee_{i\in\{7\ldots10\}}(Q_i = 4)$.

4.3. Expressing Reification and Meta-Constraints

We discuss here the means by which each language can be used to express meta-constraints and reified constraints. Formulas 1 and 2 are used to highlight the differences between the languages.

Oz, SICStus and IF/Prolog

- These languages allow a reified constraint form and admit propagators with concatenation in the way of $arg_0\ R_1\ arg_1\ R_2 \ldots R_{n-1}\ arg_{n-1}\ R_n\ arg_n$, where each R_i is a propagator and arg_{i-1} and arg_i are arguments ($i \in \{1 \ldots n\}$). For instance, the disjunction propagator $\vee$ can be concatenated in the way of $D_1 \vee D_2 \vee D_3 \vee D_4 \vee D_5$ where each D_i ($1 \leq i \leq 5$) is a Boolean variable.
- Propagators can operate directly over Boolean variables (i.e. $B_1 \vee B_2$ or $\sim B_1$ where B_1 and B_2 are Boolean variables and $\vee$ and $\sim$ represent the disjunction and negation propagators respectively[2]) or over constraint expressions (i.e. $(Q_1 = n_1) \vee (Q_2 = n_2)$ where Q_1 and Q_2 are FD variables and the values n_1 and n_2 belong to their domains).
- Meta-constraints can be combined with reified constraints as for instance in $B \equiv (Q_1 = n_1) \wedge (Q_2 = n_2)$ where B is a Boolean variable.

ECL^{i}PS^e and Ilog SOLVER

- The propagators can be concatenated as in Oz, SICStus or IF/Prolog. The main difference with these languages is that some propagators are limited to operate on constraint expressions over FD variables and not directly on the variables themselves. The direct application of the constraint propagators such as disjunction, conjunction and negation over the Boolean variables is not allowed. For instance, to express that the negation of a Boolean variable B must be true we must write $(B\# = 0)$ or $(B\backslash\# = 1)$ in ECL^{i}PS^e (resp. $(B == 0)$ or $!(B == 1)$ in Ilog).
- ECL^{i}PS^e does not allow the use of reified constraints.
- Ilog SOLVER allows the reification for the Boolean variables.

clp(FD) and B-Prolog

- Most of the FD constraint propagators have a relational form (that is, in the way of $R(X, Y, Result)$, where R is a propagator and X, Y and $Result$ are FD Boolean variables) close to the style of traditional Prolog. Thus it is not possible to employ the concatenation shown in other languages (which means that, in many cases, large numbers of extra variables are needed).
- The Boolean operators also require a relational form which means that the code needs a large number of extra FD variables.
- Meta-constraints cannot be implemented in clp(FD) in a direct way. For instance, the constraint imposed for the variable BfQ in formula 2 which must have the truth value of $\bigvee_{i\in\{1\ldots5\}}(Q_i = 4)$ is best expressed as follows:

$$(BfQ = 1) \equiv (Q1 = 4) \vee (Q2 = 4) \vee (Q3 = 4) \vee (Q4 = 4) \vee (Q5 = 4). \quad (3)$$

Code is now required that detects when the constraint $(BfQ = 1)$ is true. It is explained in [4] how the clp(FD) framework can be extended so that constraints such as this can be expressed using just constraints of the form *X in r*. However, as this idea is not implemented in the current version of the language, the approach in [7] for solving the magic square problem can be adopted. For instance, for the constraint (3) the predicate $'x = a \Leftrightarrow b'/3$ can be used. The call $'x = a \Leftrightarrow b'(X, A, B)$ means $X = A$ *iff* B is true (i.e. $B = 1$) and is defined:

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

where $'X_To_B'$ returns 1 if $X = A$, 0 if $X \neq A$ and $0 \ldots 1$ otherwise; and $'B_To_X'$, which is delayed until B is instantiated, yields A if $B = 1$ or else the range $0 \ldots \infty \backslash A$. These *user functions* are written in C and can accept ranges or terms as argument. In clp(FD) the constraint (3) can now be expressed as follows:

$$\begin{aligned}
&'x = a \Leftrightarrow b'(BfQ, 1, B1), \ 'x = a \Leftrightarrow b'(Q1, 4, B2), \ 'x = a \Leftrightarrow b'(Q2, 4, B3),\\
&'x = a \Leftrightarrow b'(Q3, 4, B4), \ 'x = a \Leftrightarrow b'(Q4, 4, B5), \ 'x = a \Leftrightarrow b'(Q5, 4, B6),\\
&or(B2, B3, B23), \ or(B4, B5, B45), \ or(B23, B45, B2345), \ or(B2345, B6, B1).
\end{aligned}$$

Note that, just for this one constraint, nine additional Boolean variables (B1, B2, B3, B4, B5, B6, B23, B45, B2345) are necessary.

• In B-Prolog, as for clp(FD), meta-constraints cannot be defined directly, but the reification of Boolean variables can be expressed by means of delay clauses (which avoids the need for any C code). This can be done by defining a predicate *iff*/3[3] as follows:

$$\begin{aligned}
&delay \ iff(X, Y, B) \ :- dvar(B), dvar(X) \ : \ true.\\
&delay \ iff(X, Y, B) \ :- dvar(B), dvar(Y) \ : \ true.\\
&iff(X, Y, B) \ :- integer(B) \ : \ (B =:= 1 -> \ X\# = Y; \ X\#\backslash = Y).\\
&iff(X, Y, B) \ :- X =:= Y \ : \ B = 1.\\
&iff(X, Y, B) \ :- true \ : \ B = 0. \qquad (4)
\end{aligned}$$

Note that #= and #\ are the equality and disequality propagators respectively. The part of the body of the clauses before the : is called a guard. The first two clauses are called *delay* clauses which have guards of the form $dvar(Z)$ which checks that its argument Z is a domain variable. As a result of the $delay/1$ clauses, the *iff*/3 clauses are only executed when either B or X and Y are instantiated to unique values. The remaining three clauses define *iff*/3. These use guards $integer(B)$, X=:=Y and $true$. Once the guard of a clause has succeeded, the remaining clauses defining *iff*/3 are discarded and the call becomes determinate. The call *iff(X,Y,B)* means $(X = Y) \Leftrightarrow B$, that is, if B is true, then the constraint X#= Y is imposed and, if B is false, the disequality $X\backslash\# = Y$ is imposed. On the other hand, if the constraint $X = Y$ is true (false), then B is imposed to be true (false). The constraint (3) is then coded as follows:

$$\begin{aligned}
&iff(BfQ, 1, B1), iff(Q1, 4, B2), iff(Q2, 4, B3), iff(Q3, 4, B4),\\
&iff(Q4, 4, B5), iff(Q5, 4, B6), or(B2, B3, B23),\\
&or(B4, B5, B45), or(B23, B45, B2345), or(B2345, B6, B1).
\end{aligned}$$

4.4. 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 provide the same formulation in different styles. To demonstrate this flexibility, the clp(FD) language can be used as a model for writing the formula 1 in CHR code. The different clp(FD) Boolean propagators can easily be simulated by the CHR constraints.

Meta-constraints and *reified* constraints are no problem in CHR. For instance, the formula 2 can be coded in CHR in a style close to Oz. This can be done by defining 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 have the value 1 (denoting true) in the last argument and the constraint in the first argument. Then, the formula 2 can be expressed in CHR as follows:

$$solve(BfQ \Leftrightarrow (Q1 = 4) + (Q2 = 4) + (Q3 = 4) + (Q4 = 4) + (Q5 = 4), 1),$$
$$solve(AfQ \Leftrightarrow (Q7 = 4) + (Q8 = 4) + (Q9 = 4) + (Q10 = 4), 1),$$
$$solve((Q6 = 1) \Leftrightarrow (BfQ * \sim AfQ), 1).$$

The first argument of the *solve* predicate simulates, with minor differences, the Oz code employed to program the formula 2. Note that the Oz arithmetic propagators $*$, $+$ and $\sim$ have been defined as the Boolean propagators conjunction, disjunction and negation respectively (as it was done in the Oz program for the SRQ).

5. Efficiency Compared

In this section we compare the efficiency of the eight constraint systems described in Section 2. Subsection 5.1 describes the two labeling strategies used and 5.2 gives the results for the solving of the SRQ in Figure 1 under the two different approaches described in Section 3. The efficiency study is extended to other benchmarks in Subsection 5.3 and the section concludes with an evaluation of the results.

Note that all of the systems (except CHR) provide some built-in symbolic constraints which we used in the benchmarks' code. In particular, for all systems for which it was provided, the *all_different* constraint was employed where applicable. Thus although we tried to maintain the same formulation for any given benchmark across all the systems, we did exploit all the facilities that were provided by a system to obtain the best possible results.

5.1. Labeling

It is well-known that constraint solving can be seen as a combination of two processes, constraint propagation and labeling. Constraint propagation is a procedure that reads the

constraint store and imposes constraints to it by means of constraint propagators [18]. Labeling assigns values to the domain variables (instantiation). Thus the labeling process consists of (1) choosing a variable (variable ordering) and (2) assigning to the variable a value belong to its domain (value ordering). The variable ordering and the value ordering used for the labeling can considerably influence the efficiency of the constraint solving when only one solution to the problem is required. It has little effect when the search is for all solutions. In this study, we considered two labelings, the *naive* labeling and the *first fail* labeling.

The *first fail* labeling uses a principle [14] which says that *to succeed, try first where you are the most likely to fail*. This principle recommends the choice of the *most constrained* variable which (for the finite domain) means choosing a variable with the smallest domain. A further refinement commonly used in *first fail* labeling consists of choosing, in the case when there is more than one such recommended variable, the one that appears in the greatest number of constraints. All the results in 5.2.2, 5.2.3 and 5.3.2 were obtained using the *first fail* labeling. As far as possible, we have maintained the same variable and value ordering for all the systems in the efficiency comparison.

We also used a *naive labeling*. In contrast to *first fail*, *naive labeling* is a very simple labeling which chooses the left-most of a list of variables and then selects the smallest value in its domain. Thus, *naive* labeling assures that both variable and value ordering are the same for all the systems and hence in many ways, although less efficient, is better for comparing the different systems when only one solution is required. Thus additional results for the first solution search were obtained using the *naive* labeling. The performance results for this are given in 5.3.3.

5.2. Efficiency Compared on the SRQ

In this subsection, we discuss the efficiency of the eight systems for solving the SRQ. We consider both of the formulations described in Section 3. Thus first we examine how the choice of representation affects the performance and hence the efficiency results. The performance of the systems for the SRQ are then compared, first under the 50 variables formulation and then under the 10 variables formulation.

5.2.1. Comparing the Search Trees

The hardness of this kind of puzzle seems to be in the number of choice-points that are traversed in finding a solution. For that reason, we compare the different Oz programs under the two different approaches (50 and 10 variables) giving them to a particular inference machine (which performs the search) called Explorer tool [24]. This allows the search tree to be visualised as shown in Figure 3. Here the choice nodes are denoted by circles, the failure nodes by squares and the solution nodes by diamonds. The left tree in Figure 3 shows the search tree of the original Oz program for *first solution* search in the SRQ solving using *first fail* labeling (see Subsection 5.1). This tree contains 27 nodes of which 13 are choice nodes. The right tree in Figure 3 shows, under the same conditions, the search tree

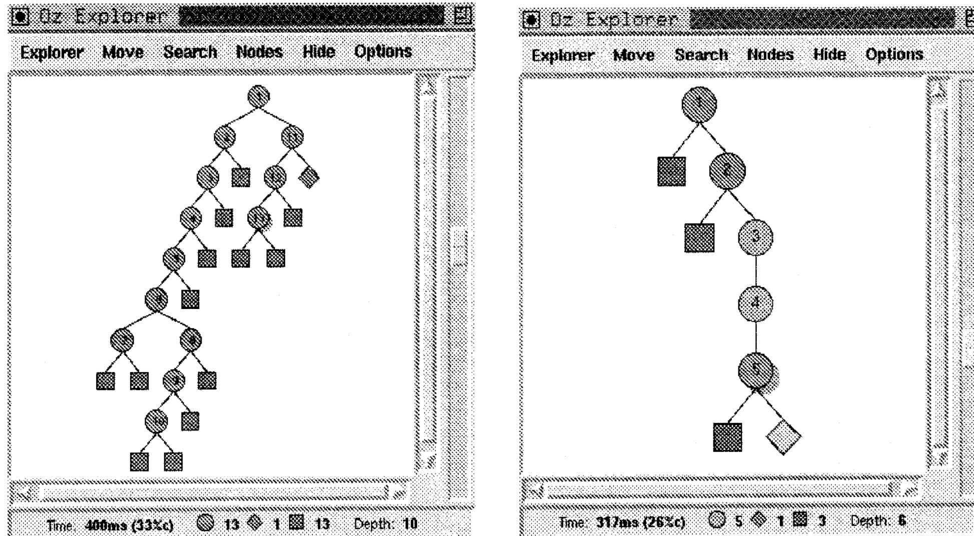

Figure 3. The Oz Explorer on SRQ solving using *first fail* labeling for *first solution* search.

for the 10 variables formulation which contains 9 nodes with only 5 choice nodes. This is more than a 60% reduction in the number of choice nodes.

5.2.2. The SRQ Results Using the Boolean Domain

Here, we present the performance of the set of programs for the SRQ implemented under the approaches shown in Subsection 3.1. The original solution in Oz involves 50 Boolean variables and all the other programs have been implemented similarly. We modified the labeling strategy for the original Oz program[4] because we wanted to do the efficiency comparisons under the same conditions of labeling. The *first fail* labeling (see Subsection 5.1) was used for all the programs except for the CHR one which used the built-in labeling for the constraint handling rules.

Since we were not able to install all the systems on the same machine, we used two machines, a 4/50 SPARCstation IPX (40 MHz) and a Pentium Pro 200 PC operating under Linux. The SRQ programs for ECL^{i}PS^e, CHR (available as a library of ECL^{i}PS^e), clp(FD), Ilog SOLVER, B-Prolog, and Oz, were measured using the SPARCstation and programs for SICStus, IF/Prolog, and again ECL^{i}PS^e, have all been run on the PC. Note that the program for ECL^{i}PS^e was measured on both machines to provide a means of comparing the results across all platforms.

Table 1 summarises the results for the programs that used the SPARCstation while Table 2 gives the results for the programs using the PC.

The meaning for the columns is as follows. The first column gives the name of the constraint language used in the implementation. The second column gives the running time

Table 1. Performance results of the 50 variables formulations for the SRQ on Sparc 40 Mhz.

language	*time first*	*time all*	*speedup*
Ilog SOLVER	80	100	3.63
clp(FD)	80	110	3.63
ECL^{i}PS^e	933	1083	↓ 3.22
CHR	6150	—	↓ 21.21
B-Prolog	217	270	1.34
Oz	290	305	1.00

Table 2. Performance results of the 50 variables formulations for the SRQ on PC (Linux).

language	*time first*	*time all*	*speedup*
SICStus	40	50	3.00
IF/Prolog	20	20	6.00
ECL^{i}PS^e	120	150	1.00

Table 3. Normalisation of the results for 50 variables formulations on the ECL^{i}PS^e column.

Ilog	*clp(FD)*	*Oz*	*CHR*	*SICStus*	*IF/Prolog*	*ECL^{i}PS^e*	*B-Prolog*
11.7	11.7	3.2	↓ 6.6	3.0	6.0	1.0	4.3

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 Table 1, this is in relation to the original program in Oz[5] whereas in Table 2 it is in relation to ECL^{i}PS^e. The symbol ↓ in Table 1 indicates that the following number is the average slow-down instead of speed-up with respect to Oz. Because the slow-down for CHR is so high we have only provided the result for the *first solution* search.

To give an idea of the efficiency of each of the systems with respect to each other, Table 3 shows the speedup of each system in relation to ECL^{i}PS^e (which is taken as reference).

These results show that for the SRQ problem, the Ilog and clp(FD) programs are fastest, while Oz, SICStus, IF/Prolog and B-Prolog were about two to four times slower. However, these latter four systems were at least three times as fast as ECL^{i}PS^e. Of course, as the times needed to solve the SRQ problem are not high, the real differences in the performance of these systems are not necessarily indicated here (in the next subsection, we compare again the performance of these systems using a larger set of benchmarks). The CHR program is

Table 4. Comparable results of the 10 variables formulation for the SRQ on Sparc 40 Mhz.

language	*time first*	*time all*	*speedup*	*speedup prec*
Ilog SOLVER	40	90	7.25	2.00
clp(FD)	30	40	9.67	2.67
ECL^{i}PS^e	583	933	↓ 2.01	1.60
CHR	4400	—	↓ 15.17	1.39
B-Prolog	33	50	8.79	6.58
Oz	186	295	1.56	1.56

Table 5. Comparable results of the 10 variables formulation for the SRQ on PC(Linux).

language	*time first*	*time all*	*speedup*	*speedup prec*
SICStus	30	40	4.00	1.33
IF/Prolog	10	10	12.00	2.00
ECL^{i}PS^e	100	140	1.20	1.20

Table 6. Normalisation of the results for 10 variables formulation on the column ECL^{i}PS^e.

Ilog	*clp(FD)*	*Oz*	*CHR*	*SICStus*	*IF/Prolog*	*ECL^{i}PS^e*	*B-Prolog*
14.6	19.4	3.1	↓ 7.5	3.3	10.0	1.0	17.7

by far the slowest. This is to be expected since the CHR used here is implemented on the top of the ECL^{i}PS^e system.

5.2.3. *The SRQ Results Using the 10 Variables Approach*

Using the 10 variables formulation in Subsection 3.2, we implemented a program for all the languages. As for the 50 variables approach the results were measured on two different machines and later normalised by the ECL^{i}PS^e results. Tables 4 and 5 show the results and Table 6 shows the speedup of each of the systems with respect to the ECL^{i}PS^e system.

The meanings for the first four columns in Tables 4 and 5 are the same as in Tables 1 and 2. Just as in the 50 variables formulation, only the result for the *first solution* search is given for CHR. Note that, in Table 4, the column *speedup* gives the speed-up with respect to the original program in Oz (that is, with respect to the 50 variables formulation in Oz) and, in Table 5, with respect to the ECL^{i}PS^e program with 50 variables. The fifth column *speedup prec* indicates the speed-up factor with respect to the 50 variables program implemented

in the same language (results in Tables 1 and 2). All programs implemented under the 10 variables approach improve the speed of the programs over those implemented in the style of the original Oz program involving 50 Boolean variables. This illustrates how a change of representation may significantly affect the performance.

Table 6 compares all the programs for the 10 variables formulation by normalising the results relative to the ECL^{i}PS^e timings.

5.3. A More Comprehensive Comparison

5.3.1. Some Extra Benchmarks

It is clear that a single benchmark may bias the performance results unfairly. Typically, few constraint languages dominate such an application and thus the resulting performance figures are vulnerable to slight variations of the implementation of these constraints. In order to make the comparison more objective we extended the work to include the following well-known benchmarks [29]:

- **sendmore**: a cryptarithmethic problem on 8 variables ranging over $0 \ldots 9$, with one linear equation and 36 disequations;
- **alpha**: a cipher problem involving 26 variables over $1 \ldots 26$, with 20 equations and 325 disequations;
- **equation 10**: a system of 10 linear equations with 7 variables over $0 \ldots 10$;
- **equation 20**: a system of 20 linear equations with 7 variables over $0 \ldots 10$;
- **N queens**: place N queens on a $N \times N$ chessboard in such a way that no queen attacks each other;
- **magic sequences (N)**: calculate a sequence of N numbers such that each of them is the number of occurrences in the series of its position in the sequence.

The programs **sendmore**, **alpha**, **equation 10** and **equation 20** test the efficiency of the systems to solve linear equation problems. The **N queens** and **magic sequences** programs are scalable and therefore useful to test how the systems works for bigger instances of the same problem. Note that both the number of variables and the number of values for each variable grow linearly with N. That is, given a value N, at least N FD variables must be declared with domains that range between 0 or 1 and N.

The programs used for the measurements for each of the extra benchmarks listed above were either provided with the system or first written by us but then improved by the language designers themselves. This policy meant that, for each system, only appropriate programs have been compared. We measured the time required both for finding just one solution and, where possible, in finding all solutions.

All the results in 5.3.2 were obtained using the *first fail* labeling. However, as observed in Subsection 5.1, the choice of labeling can affect the performance when searching for just

one solution. Thus, for just the one solution case, the systems were compared again using the *naive* labeling. The performance results for this are given in 5.3.3.

For ECL^{i}PS^e, we initially used the **N queens** program provided with the system and an adaptation of the SICStus program for the **magic sequences**. However, the performance was poor. We found that one of the reason for the inefficiency was the *first fail* labeling (see 5.3.2). When there are several with the same smallest domain, the one chosen is not defined and dependent on the implementation. In order to improve the efficiency, the *first fail* labeling of Joachim Schimpf[6] was used which was found to improve the speed by a factor of 14. To obtain further improvements, the programs were compiled without debugging information and without garbage collection[7]. As a result of this, for the **magic sequences** program, we improved the speed by a factor of 70. However, the lack of garbage collection reduced the size of problems that could be solved. It should also be noted that removing the garbage collection in other systems such as SICStus did not improve the performance.

5.3.2. The Benchmarks Measured With First Fail Labeling

All the results presented in Tables 7 to 12 were obtained using the *first fail* labeling. For each of the benchmarks, the times needed to obtain the first solution have been measured with each of the constraint systems except for CHR. For CHR, the only benchmarks used were the 50 and 10 variables solutions for the SRQ. This is because the only solutions that could be found for the other benchmarks were extremely inefficient; no solutions for these were supplied with the CHR library and running the existing ECL^{i}PS^e code with the CHR finite domain library (as suggested by the CHR author[8]) was still too slow. As an excuse for this, we observe that CHR was not built for writing efficient solvers but for defining adequate constraints solvers for particular problems on specific domains.

For the scalable problems (that is, the **N queens** and **magic sequences**), the times for finding all possible solutions have also been measured with each of the constraint systems (except, that is, for CHR).

As explained in Subsection 5.2, since we were not able to install all the systems on the same machine, measurements for clp(FD), CHR, Ilog, Oz and B-Prolog have been obtained on a different machine to SICStus and IF/Prolog. Programs for ECL^{i}PS^e were timed on both machines so that the results could be compared. All the timings are in seconds.

Tables 7 and 8 show the results with Ilog SOLVER, clp(FD), Oz, CHR, ECL^{i}PS^e, and B-Prolog, using the same SPARCstation IPC 4/40 to 25 Mhz[9]. Table 7 gives the times for finding the first solution and Table 8 the times for obtaining all solutions for the **N queens** and **magic sequences** problems.

Tables 9 and 10 show the results with SICStus, IF/Prolog, and again ECL^{i}PS^e, using a Pentium Pro 200 PC operating under LINUX. Table 9 gives the times for finding the first solution and Table 10 shows the times for obtaining all the solutions for the **N queens** and **magic sequences** problems.

In these tables, *Error1* in the clp(FD) columns means that the error message “trail stack overflow” was returned. This can be avoided by increasing the size of the environment variable associated with the stack, although the solution is very slow. The *Error2* in the

Table 7. Performance results on SParc (25 MHz) for first solution search.

Benchmark	*Ilog*	*clp(FD)*	*Oz*	*CHR*	*ECL^{i}PS^e*	*B-Prolog*
SRQ (10)	0.060	0.050	0.290	7.600	0.900	0.050
SRQ (50)	0.120	0.120	0.410	9.300	1.380	0.335
64 queens	0.110	2.710	4.190	—	10.816	0.200
100 queens	0.230	*Error1*	1.270	—	5.533	119.300
sendmore	0.010	0.020	0.020	—	0.033	0.010
alpha	0.130	0.300	0.480	—	1.633	0.233
eq. 10	0.200	0.270	0.460	—	0.833	0.367
eq. 20	0.240	0.360	0.460	—	1.050	0.817
magic(10)	0.020	0.040	0.100	—	0.333	0.284
magic(50)	0.110	1.090	1.870	—	8.234	5.300
magic(100)	0.280	4.320	16.290	—	49.700	*Error4*
magic(130)	0.410	*Error1*	35.030	—	*Error3*	*Error4*
magic(150)	0.530	*Error1*	50.790	—	*Error3*	*Error4*
magic(200)	0.860	*Error1*	*Error2*	—	*Error3*	*Error4*

Table 8. Performance results on SParc (25 MHz) for all solutions search.

Benchmark	*Ilog*	*clp(FD)*	*Oz*	*ECL^{i}PS^e*	*B-Prolog*
8 queens	0.340	0.370	2.210	2.167	0.200
9 queens	1.350	1.410	10.200	8.950	0.817
10 queens	5.350	5.270	36.540	34.467	3.216
11 queens	23.920	23.970	178.240	166.583	14.700
12 queens	118.200	119.760	836.891	840.650	72.533
magic(30)	0.160	0.890	0.890	5.050	6.300
magic(75)	1.300	8.680	9.860	55.850	102.734
magic(100)	2.690	18.380	24.580	125.600	*Error4*
magic(130)	5.300	*Error1*	48.590	*Error3*	*Error4*
magic(150)	7.640	*Error1*	83.780	*Error3*	*Error4*

Table 9. Performance results on PC (Linux) for first solution search.

Benchmark	*SICStus*	*IF/Prolog*	*ECL^{i}PS^e*
SRQ (10)	0.030	0.010	0.100
SRQ (50)	0.040	0.020	0.120
64 queens	0.500	0.300	0.820
100 queens	0.460	0.430	0.460
sendmore	0.005	0.005	0.010
alpha	0.065	8.090	0.120
eq. 10	0.045	0.010	0.055
eq. 20	0.050	0.010	0.070
magic(10)	0.020	0.020	0.030
magic(50)	0.280	0.290	0.650
magic(100)	1.080	1.570	3.830
magic(130)	1.820	2.910	7.750
magic(150)	2.330	3.700	*Error3*
magic(200)	4.180	13.450	*Error3*

Table 10. Performance results on PC (Linux) for all solutions search.

Benchmark	*SICStus*	*IF/Prolog*	*ECL^{i}PS^e*
8 queens	0.130	0.040	0.180
9 queens	0.510	0.180	0.740
10 queens	2.000	0.700	2.950
11 queens	8.810	3.150	13.340
12 queens	43.120	15.580	66.150
magic(30)	0.160	0.140	0.420
magic(75)	1.050	1.140	4.140
magic(100)	1.930	2.490	9.100
magic(130)	3.360	4.720	*Error3*
magic(150)	4.600	7.050	*Error3*

Table 11. Normalisation table for first solution search.

Benchmark	*Ilog*	*clp(fd)*	*Oz*	*CHR*	*SICStus*	*IF/Prolog*	*ECL^iPS^e*	*B-Prolog*
SRQ (10)	15.0	18.0	3.1	↓ 8.4	3.3	10.0	1.0	18.0
SRQ (50)	11.5	11.5	3.4	↓ 6.7	3.0	6.0	1.0	4.1
64 queens	98.3	4.00	2.6	—	1.6	2.7	1.0	54.1
100 queens	24.1	*Error1*	4.4	—	1.0	1.1	1.0	↓ 21.6
sendmore	3.3	1.7	1.7	—	2.0	2.0	1.0	3.3
alpha	12.6	5.4	3.4	—	1.8	↓ 67.4	1.0	7.0
eq. 10	4.2	3.1	1.8	—	1.2	5.5	1.0	2.3
eq. 20	4.4	2.9	2.3	—	1.4	7.0	1.0	1.3
magic(10)	16.7	8.3	3.3	—	1.5	1.5	1.0	1.2
magic(50)	74.9	7.6	4.4	—	2.3	2.2	1.0	1.56
magic(100)	177.5	11.5	3.1	—	3.6	2.4	1.0	*Error4*
magic(130)	216.0	*Error1*	2.5	—	4.3	2.7	1.0	*Error4*
magic(150)	87.1	*Error1*	↓ 1.1	—	1.6	1.0	*Error3*	*Error4*
magic(200)	195.0	*Error1*	*Error2*	—	3.2	1.0	*Error3*	*Error4*

Table 12. Normalisation table for all solutions search.

Benchmark	*Ilog*	*clp(fd)*	*Oz*	*SICStus*	*IF/Prolog*	*ECL^iPS^e*	*B-Prolog*
8 queens	6.4	5.9	1.0	1.4	4.5	1.0	10.9
9 queens	6.6	6.3	↓ 1.1	1.5	4.1	1.0	11.0
10 queens	6.4	6.5	↓ 1.1	1.5	4.2	1.0	10.7
11 queens	7.0	6.9	↓ 1.1	1.5	4.2	1.0	11.3
12 queens	7.1	7.0	1.0	1.5	4.2	1.0	11.6
magic(30)	31.6	5.7	5.7	2.6	3.0	1.0	↓ 1.2
magic(75)	43.0	6.4	5.7	3.9	3.6	1.0	↓ 1.8
magic(100)	46.7	6.8	5.1	4.7	3.7	1.0	*Error4*
magic(130)	11.1	*Error1*	1.2	1.4	1.0	*Error3*	*Error4*
magic(150)	11.5	*Error1*	1.1	1.5	1.0	*Error3*	*Error4*

Table 13. Performance results on SParc (25 MHz) for first solution search and *naive* labeling.

Benchmark	*Ilog*	*clp(FD)*	*Oz*	*CHR*	*ECL^{i}PS^e*	*B-Prolog*
SRQ (10)	0.060	0.050	0.290	7.600	0.890	0.050
SRQ (50)	0.120	0.120	0.405	9.300	1.383	0.350
sendmore	0.020	0.020	0.020	—	0.033	0.010
alpha	10.070	14.400	48.99	—	309.200	46.866
eq. 10	0.180	0.220	0.390	—	0.650	0.300
eq. 20	0.230	0.360	0.570	—	1.367	0.834
magic(10)	0.040	0.080	0.150	—	0.433	0.733
magic(50)	0.790	5.970	4.660	—	32.833	599.550
magic(100)	3.930	43.830	37.300	—	232.733	*Error4*
magic(130)	7.490	*Error1*	60.900	—	*Error3*	*Error4*
magic(150)	10.720	*Error1*	97.516	—	*Error3*	*Error4*
magic(200)	22.880	*Error1*	*Error2*	—	*Error3*	*Error4*

Oz column is because Oz system "died" before solving the problem. The *Error3* in the ECL^{i}PS^e columns means that there was a stack error. *Error4* in the B-Prolog column means a *Trail* or a *Control Stack Overflow* error was received. The anomalous results in Tables 7 and 9 where, for Oz, SICStus and ECL^{i}PS^e, **64 queens** took longer than the **100 queens** are due to the choice of *first fail* labeling. The results shown in 5.3.3 using the *naive* labeling are more consistent.

The results shown in Tables 7 to 10 were normalised by means of the ECL^{i}PS^e timings. The normalisation of results for first solution search is shown in Table 11 and for all solutions search is shown in Table 12. Each cell contains the speedup with respect to the ECL^{i}PS^e solution. Again the symbol ↓ means that the following number is the average slow-down with respect to ECL^{i}PS^e system. In the cases in which ECL^{i}PS^e returned an error, we calculated the average differences between the two machines used to measure the results and normalised the results for IF/Prolog. Note that this only occurs in the last two rows of Tables 11 and 12.

5.3.3. The Benchmarks Measured With Naive Labeling

The efficiency results using the *naive* labeling (see Subsection 5.1) are shown in Tables 13, 14 and 15. Note that, for the **N queens** problem, no results are shown because the running times were too high. Only results for first solution search are shown. The results for all solutions search were similar to those shown using the *first fail* labeling.

Table 14. Performance results on PC (Linux) for first solution search and *naive* labeling.

Benchmark	*SICStus*	*IF/Prolog*	*ECL^iPS^e*
SRQ (10)	0.030	0.010	0.100
SRQ (50)	0.040	0.020	0.120
sendmore	0.005	0.005	0.010
alpha	5.930	51.280	23.360
eq. 10	0.035	0.010	0.045
eq. 20	0.060	0.010	0.090
magic(10)	0.020	0.020	0.040
magic(50)	0.710	0.540	2.180
magic(100)	2.910	3.170	14.190
magic(130)	5.130	6.480	31.400
magic(150)	7.150	9.140	*Error3*
magic(200)	13.350	25.410	*Error3*

Table 15. Normalisation table for first solution search and *naive* labeling.

Benchmark	*Ilog*	*clp(fd)*	*Oz*	*CHR*	*SICStus*	*IF/Prolog*	*ECL^iPS^e*	*B-Prolog*
SRQ (10)	14.8	17.8	3.1	↓ 8.5	3.3	10.0	1.0	17.8
SRQ (50)	11.5	11.5	3.4	↓ 6.7	3.0	6.0	1.0	4.0
sendmore	1.7	1.7	1.7	—	2.0	2.0	1.0	3.4
alpha	30.7	21.5	6.3	—	3.9	↓ 2.2	1.0	6.6
eq. 10	3.6	3.0	1.7	—	1.3	4.5	1.0	2.2
eq. 20	5.9	3.8	2.4	—	1.5	9.0	1.0	1.6
magic(10)	10.8	5.4	2.9	—	2.0	2.0	1.0	↓ 1.7
magic(50)	41.6	5.5	7.0	—	3.1	4.0	1.0	↓ 18.3
magic(100)	59.2	5.3	6.2	—	4.9	4.5	1.0	*Error4*
magic(130)	52.3	*Error1*	6.4	—	6.1	4.8	1.0	*Error4*
magic(150)	10.6	*Error1*	1.2	—	1.3	1.0	*Error3*	*Error4*
magic(200)	13.8	*Error1*	*Error2*	—	1.9	1.0	*Error3*	*Error4*

Table 16. Number of FD variables managed in the **magic sequences** problem.

Ilog	*clp(fd)*	*Oz*	*ECL^{i}PS^e*	*B-Prolog*
1624–1625	111–112	194–200	230–???	82–83

Table 17. Number of FD variables managed in the **magic sequences** problem.

ECL^{i}PS^e	*SICStus*	*IF/Prolog*
228–???	1000–1200	300–600

5.3.4. Robustness

Using the same machines as for the efficiency comparison in 5.3.2 and 5.3.3, the robustness of each the systems was measured. For this, we used the **magic sequences (N)** programs (with garbage collection on) and measured the maximum value of **N** that each system could manage. Tables 16 and 17 give an interval of FD variables. Each system succeeded at the minimum of the interval while it failed at the maximum. Note that, for both machines, there was no upper bound for ECL^{i}PS^e. This was due to the garbage collection which made these tests for ECL^{i}PS^e extremely slow. CHR was not evaluated. In spite of the different configurations of the two machines used, the results on them for ECL^{i}PS^e are almost the same.

Note that the precise values in these results are very dependent on the machine used. However, these results do provide some indication of the comparative robustness of the systems.

5.4. The Results Analysed

In this section we summarise and compare the performance results provided in Subsections 5.2 and 5.3.

Ilog SOLVER. In general, Ilog was by far the fastest system. Ilog was also extremely robust, solving the **magic sequences** problem with over 1600 variables.

clp(FD). This gave good results, although it was not as fast as Ilog. Unfortunately, it gave error messages when the problem size was increased, indicating that it does not scale well with respect to the number of FD variables. We have been able to solve larger problems by changing the size of certain environment variables but the performance was really poor. For example, with such a change, clp(FD) solved the **100 queens** problem for first solution search in 126 seconds (almost 23 times slower than ECL^{i}PS^e).

Oz

Oz was faster than ECL^{i}PS^e when finding the first and all solutions. When obtaining all solutions for the **magic sequences** problem, it was almost as fast as clp(FD). Also, Oz was more robust than clp(FD) and only failed to obtain a solution for the **magic sequences (200)** problem.

SICStus and IF/Prolog

These had very similar performance figures and were about two to three times as fast as ECL^{i}PS^e (although IF/Prolog performed badly with the **alpha** benchmark). Note that IF/Prolog worked particularly well for first solution search (even sometimes better than clp(FD) and Ilog). SICStus and IF/Prolog were also more robust than clp(FD), Oz, and ECL^{i}PS^e. However, of the two, SICStus has the greater robustness since it was able to solve the **magic sequences** problem with over 1000 FD variables whereas IF/Prolog failed to solve the same problem with 600 FD variables.

B-Prolog

As for clp(FD), this system worked well with problems involving a small number of FD variables. However, as we increased the number of FD variables for the **queens** and **magic sequences** problems, performance deteriorated rapidly leading, in most cases, to the program being aborted with error messages. This was a direct consequence of the fact that this version of B-Prolog does not have a garbage collector.

ECL^{i}PS^e

This had the slowest results (except for CHR). To obtain the best possible performance, the figures in Table 7 for **ECL^{i}PS^e** had the garbage collection disabled. With garbage collection, the **magic sequences (100)** problem took three times longer to find a solution. We had no results and an error message for the scalable benchmarks when the number of FD variables was large. In fact, in many cases these problems could be solved but with a very poor performance. For example, without garbage collection, the **magic sequences (N)** problem with $\mathbf{N} \geq 130$ could not be solved. However, when the garbage collection was enabled, it was solved (on the Sparc Station) for first solution search in 480 seconds for $\mathbf{N} = 130$, 795 seconds for $\mathbf{N} = 150$ and 1977 seconds for $\mathbf{N} = 200$.

CHR

This was slowest. Two reasons for this: (1) the system we tested is built on top of ECL^{i}PS^e (which had poor performance) and (2) CHR was not designed primarily for efficiency but for defining adequate constraints solvers for particular problems on specific domains.

6. Conclusions and Further Work

In this paper, eight popular but very different constraint systems under two different approaches (*black box*, *glass box*) have been compared on the Boolean and finite domains. We focused the comparison on the efficiency and on specific aspects of the expressivity (concerning reified constraints and meta-constraints). By showing the main differences between the systems, we have provided some guidelines that should help the choice of an adequate constraint language for solving a specific constraint satisfaction problem.

It should be noted, that our conclusions are based on simple problems. Implementing these in the eight systems required a considerable amount of work and the tests needed a large amount of computing time and power. More resources are needed if more realistic problems are to be used in comparing and contrasting the CLP systems. It should be also noted that we have only compared the FD libraries. Similar comparative studies could also be made for other common domains such as Intervals [2] or Reals [19]. Alternative approaches such as integer linear programming could also be considered for a comparison between systems. This could be the subject of further work.

To summarise our results, for maximum efficiency Ilog SOLVER is best. clp(FD) is also a good candidate provided the size of the problem (measured in number of FD variables) is fairly small. From an expressive point of view, CHR is best. This supports the accepted view that CHR's are particularly useful for building specialised constraint solvers for non-standard applications. Finally, for a balance between the expressiveness and efficiency, IF/Prolog and SICStus both do well although, in the tests here, SICStus had the greater robustness.

There is no language which has an expressiveness of CHR but efficiency comparable with Ilog SOLVER. Thus, an implementation of a CHR solver on the top of an efficient system such as Ilog might be worthwhile. Alternatively, it would be useful if the two different *glass box* approaches, indexicals and CHR, could be combined to give a new *glass box* approach with the benefits of both clp(FD) and CHR.

The SRQ used in this study combines a number of interesting features. Thus these puzzles can themselves provide useful benchmarks for evaluating new implementations of existing FD languages as well as a basis for studying new language extensions. All the SRQ solutions compared here are available over Internet [12].

Acknowledgments

We are grateful to the many people who have helped us with this work. In particular, we would like to thank Daniel Diaz for answering our questions on clp(FD); Gyuri Lajos and Mark Wallace for their useful comments; Joachim Schimpf, and the users of the Oz and ECL^{i}PS^e systems for their help in improving our solutions and Mats Carlsson for his SICStus solution for the **magic sequences** problem, his useful comments about SICStus and for sending the newer clp(FD) library for SICStus. We also would like to thank Jörg Würtz for providing us with efficient Oz solutions to the **magic sequences** and **N queens** problems and for his helpful comments in a preliminary version of this paper. We also wish to thank Andrew Verden for answering our questions about IF/Prolog and Helmut

Simonis for sending us some data and solution files although they were not used for the comparison. We are also grateful to the anonymous referees for all the useful comments on earlier versions of this paper.

Finally, we would like to thank EPSRC (grants GR/L19515 and GR/M05645) and CICYT (grant TIC98-0445-C03-03) for partly supporting different stages of this research.

Notes

1. Such as that suggested by Mark Wallace (personal communication), now available in [12].
2. The syntax of the propagator changes for each language.
3. Personal communication with Neng-Fa Zhou.
4. 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.
5. Note that Oz has been taken as reference since it provided the original solution.
6. Personal communication with Joachim Schimpf.
7. Personal communication with Mark Wallace and Joachim Schipmf.
8. Personal communication with Thom Frühwirth.
9. Note this machine is different to that used to measure results on SRQ solving in Subsection 5.2.

References

1. ECL^{i}PS^e 3.5, User Manual. (1995). ECRC, Munich.
2. Benhamou, F. (1994). Interval constraint logic programming. In *Constraint Programming: Basics and Trends*, (A. Podelski, editor), *LNCS 910*, Springer Verlag, pp. 1–21.
3. Carlsson, M., Ottosson, G. & Carlson, B. (1997). An open-ended finite domain constraint solver. *Proc. of the 9th International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'97)*, *LNCS 1292*, Springer Verlag, pp. 191–206.
4. Carlson, B., Carlsson, M. & Diaz, D. (1994). Entailment of finite domain constraints. *Proc. of the 11th International Conference on Logic Programming*, MIT Press, pp. 339–353.
5. Codognet, P. & Diaz, D. (1993). A minimal extension of the WAM for clp(FD). *Proc. of the 10th International Conference on Logic Programming*, MIT Press, pp. 774–790.
6. Codognet, P. & Diaz, D. (1994). clp(B): Combining simplicity and efficiency in Boolean constraint solving. *Proc. of the 6th International Symposium on Programming Languages Implementation and Logic Programming (PLILP'94)*, *LNCS 844*, Springer Verlag, pp. 244–260.
7. Codognet, P. & Diaz, D. (1996a). Compiling constraints in *clp(FD)*. *The Journal of Logic Programming* 27: 185–226.
8. Codognet, P. & Diaz, D. (1996b). Local propagation methods for solving Boolean constraints in constraint logic programming. *The Journal of Automated Reasoning* 17(1).
9. Cras, J-Y. (1993). A review of industrial constraints solving tools. *AI Intelligence*.
10. Csontó, J. & Paralič, J. (1997). A look at CLP: theory and application. *Applied Artificial Intelligence* 11: 59–69.
11. Fernández, A. J. & Hill, P. M. (1997). Boolean and finite domain solvers compared using self referential quizzes. *Proc. of the Joint Conference on Declarative Programming (APPIA-GULP-PRODE'97)*, pp. 533–544.

12. Fernández, A. J. (1998). $http://www.lcc.uma.es/\sim afdez/srq$.

13. Frühwirth, T. (1994). Constraint handling rules. In *Constraint Programming: Basics and Trends*, (A. Podelski, editor), *LNCS 910*, Springer Verlag, pp. 90–107.

14. Haralick, R. & Elliot, G. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence* 14: 263–313.

15. Henz, M. (1996). Don't be puzzled! *Workshop on Constraint Programming* in conjunction with the *2nd International Conference on Principles and Practice of Constraint Programming (CP'96)*.

16. IF/Prolog V5.0A, constraints package. (1994). Siemens Nixdorf Informationssysteme AG, Munich, Germany.

17. Ilog SOLVER, Reference Manual, version 3.1. (1995).

18. Jaffar, J. & Lassez, J. L. (1987). Constraint logic programming. *Proc. of the 14th ACM Symposium on Principles of Programming Languages (POPL'87)*, pp. 111–119.

19. Jaffar, J., Michaylov, S., Stuckey, P. & Yap, R. (1992). The CLP($\Re$) language and system. *ACM Transactions on Programming Languages and Systems* 14(3): 339–395.

20. Jaffar, J. & Maher, M. J. (1994). Constraint logic programming: a survey. *The Journal of Logic Programming* 19&20: 503–581.

21. Müller, T. & Würtz, J. (1996). Interfacing propagators with a concurrent constraint language. *Workshop on Parallelism and Implementation Technologies for (Constraint) Logic Languages*.

22. N'Dong, S. (1997). Prolog IV ou la programmation par contraintes selon PrologIA. *Proc. of Sixièmes Journées Francophones de Programmation Logique et Programmation par Contraintes (JFPLC'97)*, pp. 235–238.

23. Puget, J-F. & Leconte, M. (1995) Beyond the glass box: constraints as objects. *Proc. of International Symposium on Logic Programming (ILPS'95)*, MIT Press.

24. Schulte, C. (1995). Solver–An Oz search debugger. *Proceedings of International Workshop on Oz Programming (WOz'95)*, Institut Dalle Molle d'Intelligence Artificielle Perceptive, Martigny, Switzerland.

25. SICStus Prolog User's manual, release 3#5. (1996). By the Intelligent Systems Laboratory, Swedish Institute of Computer Science.

26. Sidebottom, G. (1993). A Language for Optimizing Constraint Propagation. PhD Thesis, Simon Fraser University.

27. Smolka, G. (1995). The Oz programming model. In *Computer Science Today*, (Jan van Leeuwen, editor), *LNCS 1000*, Springer Verlag, pp. 324–343.

28. Van Hentenryck, P. (1988). Tutorial on the CHIP systems and applications. *Workshop of Constraint Logic Programming*. Rehovot, Israel, Weizmann Institute of Science.

29. Van Hentenryck, P. (1989). Constraint Satisfaction in Logic Programming. MIT Press.

30. Van Hentenryck, P., Saraswat, V. A. & Deville, Y. (1994). Design, implementation and evaluation of the constraint language cc(FD). In *Constraint Programming: Basics and Trends*, (A. Podelski, editor), *LNCS 910*, Springer Verlag, pp. 293–316.

31. *Proc. of the Sixth International Conference on the Practical Application of Prolog and the Fourth International Conference on the Practical Application of Constraint Technology (PAPPACT98)*. (1998). Publisher Practical Application Company Ltd.

32. Zhou, N-F. (1997). B-Prolog User's Manual (Version 2.1). Faculty of Computer Science and Systems Engineering, Kyushu Institute of Technology, Fukuoka, Japan.