A Study to Design Computer Aided Programming System

Avtar Singh

Research Scholar, Manav Bharti University, Himachal Pradesh, India

ABSTRACT: This paper describes a computer-aided constraint programming system. Traditional Constraint Programming Languages have been built on top of host languages such as Prolog, Lisp, C++. This means that the user must have reasonable knowledge of the syntax and semantics of the host language before being able to use the constraint technology effectively. On top of this, the user may also be required to specify the heuristics and, or algorithm to solve the constraint problem. This leads to a bottleneck in the amount of people who have the necessary expertise in both constraint programming and the host language to implement practical systems, which use constraint satisfaction techniques. Our aim is to abstract out as many of these details as possible, to produce a high level system, where the problem specification is the focus. We have defined a simple, intuitive, high level, declarative (the order in which constraints are specified has no significance) language called EaCL for specifying constraint satisfaction problems. We propose an open architecture in which future constraint solvers can reside. The architecture also allows multiple flexible interfaces. In this paper we present as an example, an exam time tabling system built on top of our system, using Visual Basic and Automation.

---♦-------------------------------------

1. INTRODUCTION

A constraint satisfaction problem is a problem where one is given a finite set of variables, each of which is associated with a (normally finite) domain. Constraints restrict the values to be taken by the variables simultaneously. The problem is to assign a value to each variable satisfying all the constraints [14],[3],[5].

Constraint programming systems have had remarkable achievement in many applications. Many more applications could have benefited from it had there been more experts in the field to exploit the technology. Successful though they are, previous approaches to building constraintprogramming systems have been based on taking some host language, e.g. C++ (e.g. ILOG solver [10]), Lisp (e.g. PECOS [11]) or Prolog (e.g. ECLiPSE [7], CHIP [12], the CHIC 2 project [2]), augmented in some way with constraint technology. This means that the user of these constraint programming systems needs to have two basic skills before they can make use of the traditional constraint programming systems:

- Be able to formulate the problem as a constraint satisfaction problem,
- Be able to program in the host language.

Some recent global optimisation modelling languages, e.g. HELIOS, ILOG Numerica [8],[15] allow users to define their problems, mathematically, almost as they would in technical papers.

Our aim is to minimise the amount of knowledge required by the end user to be able to start using our system. Our approach is to use a high level language similar in some ways to HELIOS and ILOG Numerica, but targeted at

Constraint Satisfaction Problems, while still maintaining the ability of our system to be used for practical applications

2. ARCHITECTURE

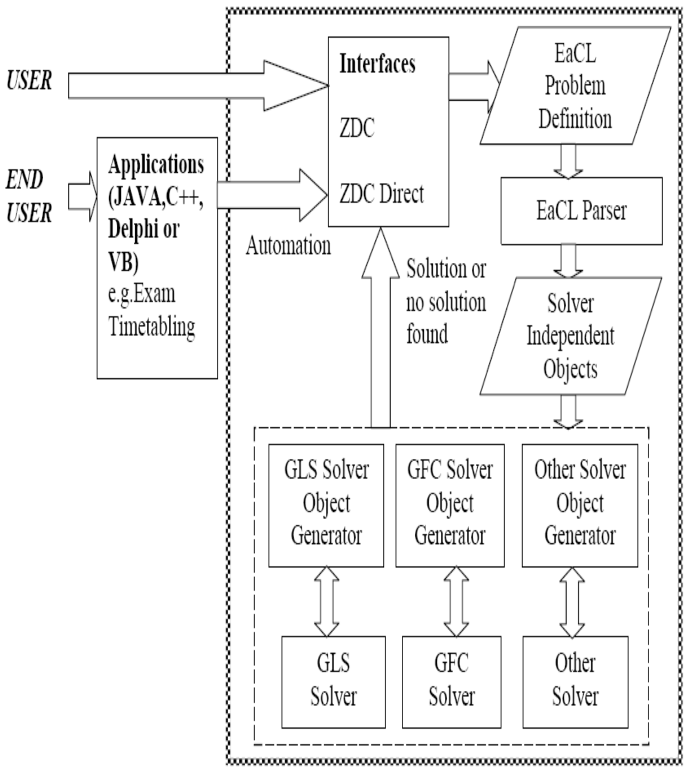

USER
END USER
Applications (JAVA,C++, Delphi or VB) e.g.Exam Timetabling
Interfaces
ZDC
ZDC Direct
Automation
EaCL Problem Definition
EaCL Parser
Solution or no solution found
Solver Independent Objects
GLS Solver Object Generator
GFC Solver Object Generator
Other Solver Object Generator
GLS Solver
GFC Solver
Other Solver

Figure 1: The CACP Architecture

At the top level our current implementation supports two user interfaces for entering constraint satisfaction problems, ZDC (see Figure 2) and ZDCDirect (see Figure 3). ZDCDirect allows direct entry of the problem, using the EaCL without any special graphical user interfaces.

ZDC contains a formulation wizard, domain, variable and constraint builders, problem browser and online tutorials and examples, all aimed at easing the problem formulation process. In addition to this, both of these can be used as Automation Servers (automation is a technique which allows objects to make functions and data available to other objects or applications [13]), allowing real applications to be built on top of ZDC or ZDCDirect. This could be done using Visual Basic, Visual C++ or JAVA, or from another application such as ACCESS, EXCEL or even WORD. See Section 4 for an example

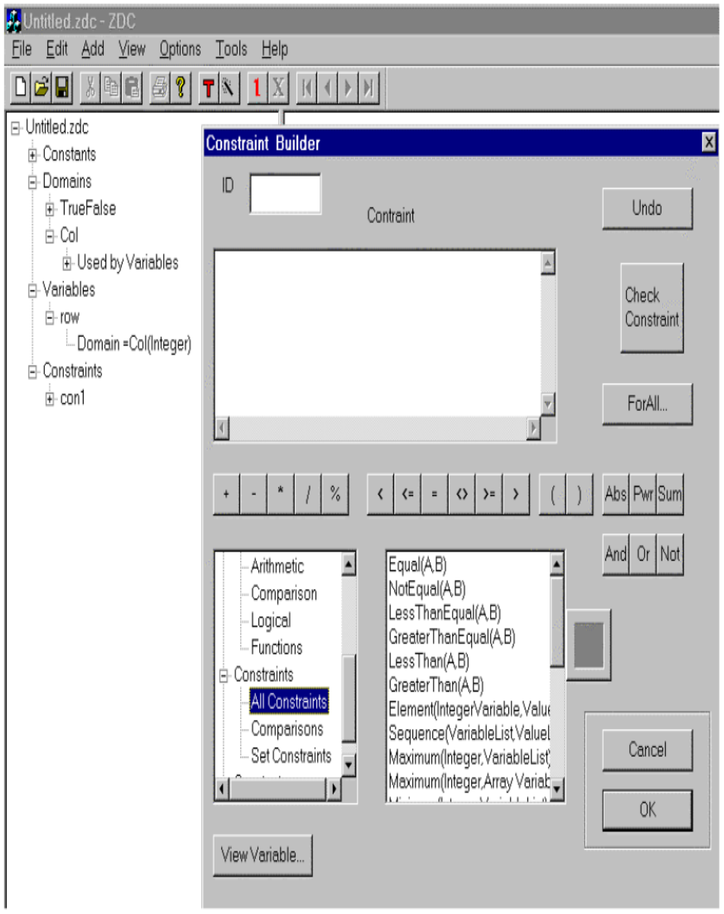
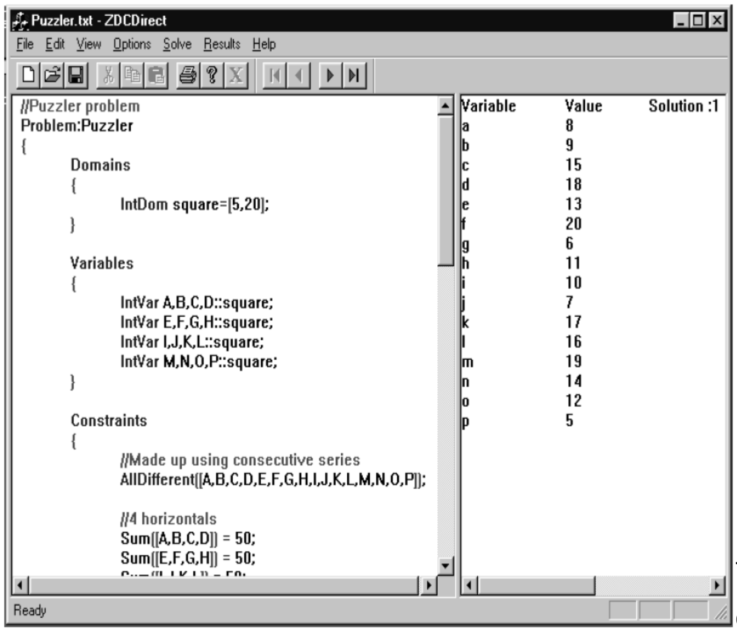


Figure 2: The ZDC interface and constraint builder

Both ZDC (Figure 2) and ZDCDirect (Figure 3) use EaCL, a high level declarative language, as their core language for describing CSPs. Once a problem has been formulated in EaCL using either interface, its syntax and semantics are checked to ensure they are correct and, if not, the interface will return a error message indicating where the error occurred and what it might be. Then the EaCL is translated into a solver-independent representation, which can then be translated by a solver object generator (one for each solver in the system) into solver dependent objects, and solved by that particular solver, with the solution returned by the interface (the user may also find all solutions, or a maximum number of solutions etc.). This makes it very easy to incorporate a new solver into our system, since all that is required is an object generator to be built which translates the solver independent objects into the new solver's representation.

Thus we have a very open architecture, because addition of a new solver requires no modification of the top level parser or language, and only knowledge of the solver-independent objects and the new solver. This means that a well-written third party solver could be incorporated into our architecture with minimum effort, as long as it supports the constraints in our language. So far, we have implemented two solvers, one based on the Forward Checking algorithm [4] (a complete algorithm) and another based on Guided Local Search [17] (an incomplete algorithm), both generalized to handle EaCL.



```
//Puzzler problem
Problem:Puzzler
{
        Domains
        {
                IntDom square=[5,20];
        }

        Variables
        {
                IntVar A,B,C,D::square;
                IntVar E,F,G,H::square;
                IntVar I,J,K,L::square;
                IntVar M,N,O,P::square;
        }

        Constraints
        {
                //Made up using consecutive series
                AllDifferent([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]);

                //4 horizontals
                Sum([A,B,C,D]) = 50;
                Sum([E,F,G,H]) = 50;
```

```
Variable     Value     Solution :1
a            8
b            9
c            15
d            18
e            13
f            20
g            6
h            11
i            10
j            7
k            17
l            16
m            19
n            14
o            12
p            5
```

Figure 3: The ZDCDirect Interface

3. THE EaCL LANGUAGE

Here we give a brief description of the EaCL language Version 1.0, which forms the core of our system (for a full description see [9], available through http://cswww.essex.ac.uk/CSP/cacp.html).

The problem file for EaCL 1.0 is split into four subsections (see Figure 4).

```
Problem:TheProblemName
{
     Data
     {
          //Constant data relevant to a particular problem
     }

     Domains
     {
          //Domain declarations
     }

     Variables
     {
          //Variable declarations
     }

     Constraints
     {
          //Constraint declarations
     }
}
```

Figure 4: The skeleton of an EaCL file

The data section can be used to store named constant data, which will typically define an instance of a particular problem. For example, it may contain a named list of lists defining what exam which student takes, etc. The domains section defines named sets of values which a variable can take, and the variables section declares the variables present in the problem, together with the name of their domain, with the constraints section defining the constraints on the variables for the problem.

The EaCL language Version 1.0 allows for three types of variables: boolean variables, integer variables and set variables. Integer and set variables must have their domains specified in the domains section, whilst boolean variables obviously only have one possible domain (0,1). Below is a list of constraints, functions and operators, which can be used to form constraints in EaCL 1.0:

Logical: And, Or, Xor, Not, Iff, Implies

Integer: -, +, *, /, %, Abs, Power, Sum, ScalProd, Count, Minimum, Maximum, =, <>, <, >, <=, >=

Set: Member, NotMember, Subset, StrictSubset, Union, Intersection, AllDisjoint, #

Symbolic: AllDifferent, Circuit, Sequence, Element

These constraints are similar to the types of constraints found in large commercial constraint programming libraries and CLPs such as ILOG solver [10] or CHIP [12], and therefore are the kinds of constraints which are likely to be useful for building real applications. Since these may not cater for every eventuality, user-defined constraints may be added which are only expressed in terms of the constraints and functions above, and other user-defined constraints, for example:

```
Constraint AtLeastButAtMost(NMin, NMax, Vars, Vals)
{
    Count(Vars, Vals) >= NMin;
    Count(Vars, Vals) <= NMax;
}
```

In other environments, e.g. in ILOG solver, user defined constraints are sometimes defined by daemons (functions which when some event occurs, perform some action) which define how each user defined constraint is propagated, when a variable’s domain is modified. Whilst this increases the power of these kinds of user-defined constraints, it also requires the user to have a deep understanding of constraint technology.

If the user requires some other function to be defined, this can also be done in a similar way. For example:

```
Function Squared(X)
{
    return X * X;
}
```

In addition to this, Forall constructs can be used to index arrays of variables, to generate groups of similar constraints, e.g. constraints in the N- queens problem:

```
Forall (i in [0..n-1], j in [i+1..n-1])
{
    Row[i] <> Row[j];
    Row[i] - Row[j] <> i - j;
    Row[i] - Row[j] <> j - i;
}
```

The language also supports intensional lists and sets. For example one can define a constraint that sums the values of all elements of an array up to element j, and specify that it is less than a certainTotal:

```
Sum([ x[i] | i < j, i in [0..N]]) < Total;
```

In addition to these features, EaCL 1.0 also supports:

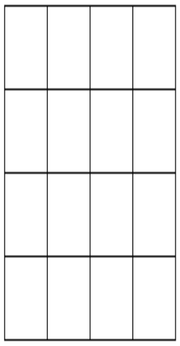
If-Else constructs on indexes for conditional definition of constraints,

Concatenation of lists and arrays, e.g. [1,2] ++ [3,4] etc.

3.1 EXAMPLE EACL FILE: THE PUZZLER PROBLEM

The puzzler problem (From Computer Weekly, 7th August 1997) is a simple example of how elegantly a problem can be specified in EaCL 1.0. It consists of a 4´4 Magic Square, which is made up using the consecutive series 5-to-20 and gives a Constant total of 50 in many different ways. The 50 total is produced by the sum of:

- 4 horizontals: ABCD, EFGH, IJKL, MNOP
- 4 verticals: AEIM, BFJN, CGKO, DHLP
- 2 long diagonals: AFKP, MJGD
- 4 three-one broken diagonals: DOJE, MBGL, ANKH, PIFC
- 2 two-two broken diagonals: CHIN, EBLO
- 9 segments: ABEF, BCFG, CDGH, EFIJ, FGJK, GHKL, IJMN, JKNO, KLOP
- 6 opposites: ABMN, BCNO, CDOP, AEDH, EIHL, IMLP
- The puzzle also specifies that P should be set to 5 and F set to 20.



As one can see below, this problem is very simple to specify using EaCL 1.0, although it only shows the basic features of EaCL 1.0. The formulation consists of 16 variables, name A to P, which must take values from the Domain square, defined to be the range of integers from 5 to 20.

Then an AllDifferent constraint specifies that all the variables should take different values (i.e. use the whole 5 to 20 range of values), and then various equality constraints define the combinations which add up to 50. Two equality constraints, setting P = 5 and F = 20 are also used.

```
Problem:Puzzler
{
    Domains
    {
        IntDom square=[5,20];
    }

    Variables
    {
        IntVar A,B,C,D::square;
        IntVar E,F,G,H::square;
        IntVar I,J,K,L::square;
        IntVar M,N,O,P::square;
    }

    Constraints
    {
        //Made up using consecutive series
        AllDifferent([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]);


                //4 horizontals
                Sum([A,B,C,D]) = 50;
                Sum([E,F,G,H]) = 50;
                Sum([I,J,K,L]) = 50;
                Sum([M,N,O,P]) = 50;

                //4 verticals
                Sum([A,E,I,M]) = 50;
                Sum([B,F,J,N]) = 50;
                Sum([C,G,K,O]) = 50;
                Sum([D,H,L,P]) = 50;

                //2 long diagonals
                Sum([A,F,K,P]) = 50;
                Sum([M,J,G,D]) = 50;

                //4 3-1 diagonals
                Sum([A,N,K,H]) = 50;
                Sum([P,I,F,C]) = 50;
                Sum([D,O,J,E]) = 50;
                Sum([M,L,G,B]) = 50;

                //2 2-2 diagonals
                Sum([I,N,H,C]) = 50;
                Sum([B,E,O,L]) = 50;
```

```
            //9 Segments
            Sum([A,B,E,F]) = 50;
            Sum([B,C,F,G]) = 50;
            Sum([C,D,G,H]) = 50;
            Sum([E,F,I,J]) = 50;
            Sum([F,G,J,K]) = 50;
            Sum([G,H,K,L]) = 50;
            Sum([I,J,M,N]) = 50;
            Sum([J,K,N,O]) = 50;
            Sum([K,L,O,P]) = 50;

            //6 opposites
            Sum([A,B,M,N]) = 50;
            Sum([B,C,N,O]) = 50;
            Sum([C,D,O,P]) = 50;
            Sum([A,E,D,H]) = 50;
            Sum([E,I,H,L]) = 50;
            Sum([I,M,L,P]) = 50;

            //Assignment constraints
            P = 5;
            F = 20;
        }
}
```

4. EXAMPLE APPLICATION: EXAM TIMETABLING

As an example of the use of our system, we solve a real world problem of exam timetabling [1]. Many Universities and schools face this type of problem, which is typically solved by hand over a period of weeks. The problem is defined as follows:

Given:

a set of slot times when exams may take place

- the default length of each exam
- the minimum time period a student must have between each exam
- a set of time slots, when specific exams must or must not take place
- which exams each student must take.

Find:

an assignment of slots to exams, such that no student is required to take exams less than the minimum time period apart, and no exam takes place in an illegal slot.

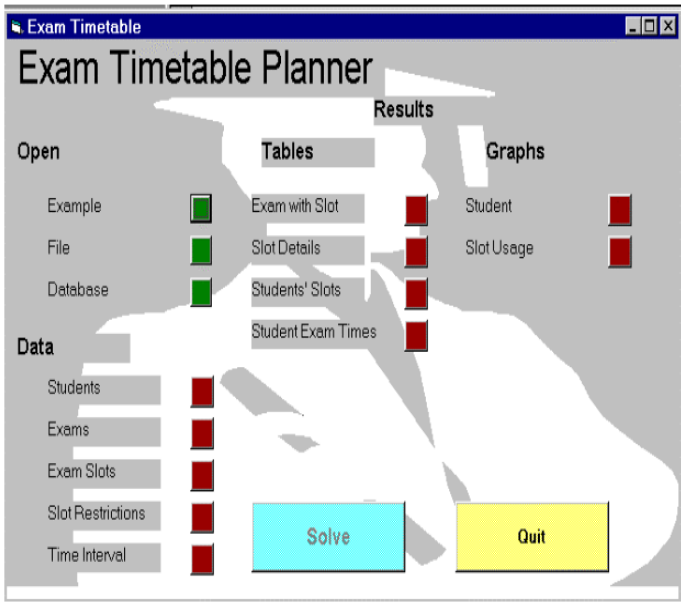


Figure 5: The Exam time tabling application built on top of ZDC

This problem can be formulated as a CSP as follows:

Variables represent the slot number when each exam takes place:

```
IntVar ExamSlots[NumExams]::Slots;
```

The domain of all variables is the set of all the possible slot numbers:

```
IntDom Slots = [0,NumSlots];
```

Constraints:

1. Some exams must not take place in certain slots, and some must take place in certain slots:

```
ExamSlot[ExamExcluded] <> ExcludedSlot;
ExamSlot[ExamMust]      =  MustSlot;
```

2. Exam containing common students, must be at least the minimum time period apart:

```
Abs(SlotTimes[ExamSlots[examIdxCommon]]-
    SlotTimes[ExamSlots[examIdxCommon]])>MinInterval;
```

(SlotTimes is an array in the data section, specifying the time in minutes when a slot for a possible exam starts each day)

Automation is used from Visual Basic to call our ZDC application, to assemble the data, domains, variables and constraints necessary to solve a particular instance of an exam time tabling problem. The details of each problem are stored in a Microsoft Access database (students and exams they take, slot times and the time interval allowed between exams).

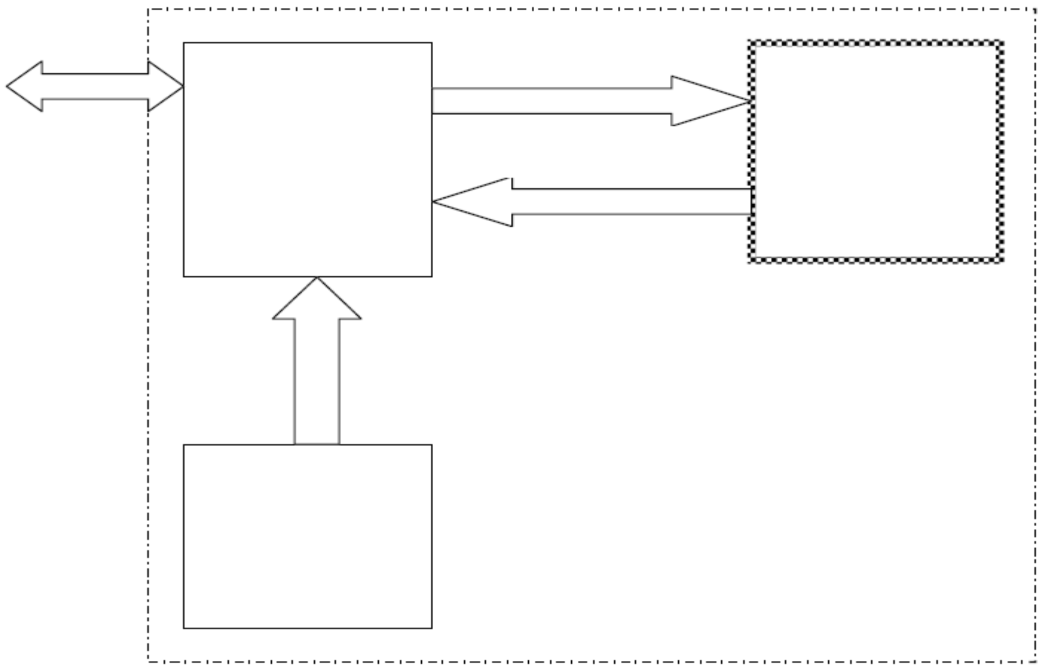


Figure 6: Data flow in the exam timetabling application, built on top of ZDC

Once the problem has been solved, the solution can be visualised in lots of different ways (these are implemented using standard Visual Basic components). For example:

Which slot each exam takes place in,

The exam timetable for each student (see Figure 7).

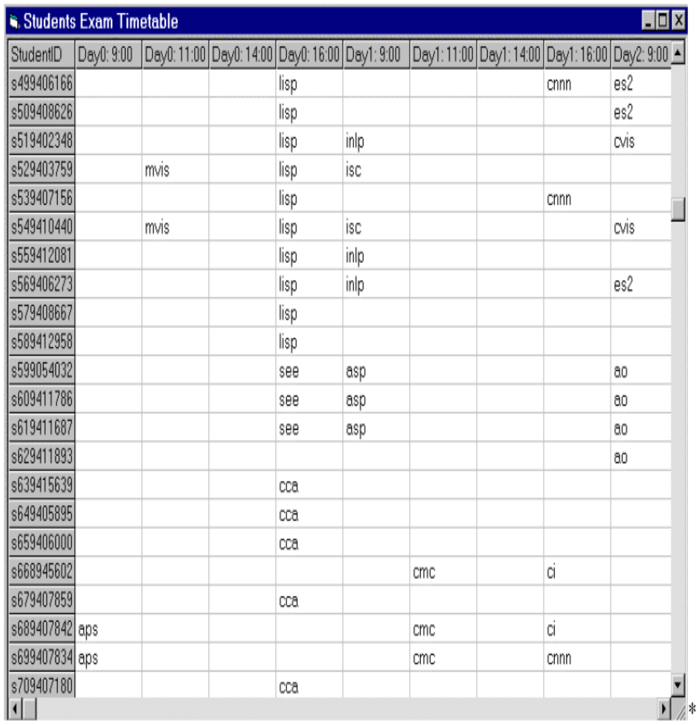
Bar chart of the number of students per slot (see Figure 8),

Bar chart of the number of exams per slot, The exams in each slot.

We have found that our system solves the exam timetabling problem adequately using either solver (although the aim of our system is not outright performance, but usuability). It required only half a day to build the constraint programming part (the basic problem formulation was developed in ZDC and then integrated with the rest of the timetabling system) of the system.

Together with a non-trivial set of statistics (to allow users to verify the results and to visualize the timetable), it took less than a week to build a practical exam timetabling system which is capable of using publicly available instances of the exam timetabling problem1.

This shows how effective and easy it is to use our system. It also shows how feasible it is to represent and solve real-world problems using our system.



StudentID	Day0: 9:00	Day0: 11:00	Day0: 14:00	Day0: 16:00	Day1: 9:00	Day1: 11:00	Day1: 14:00	Day1: 16:00	Day2: 9:00
s499406166				lisp				cnnn	es2
s509408626				lisp					es2
s519402348				lisp	inlp				cvis
s529403759		mvis		lisp	isc				
s539407156				lisp				cnnn	
s549410440		mvis		lisp	isc				cvis
s559412081				lisp	inlp				
s569406273				lisp	inlp				es2
s579408667				lisp					
s589412958				lisp					
s599054032				see	asp				ao
s609411786				see	asp				ao
s619411687				see	asp				ao
s629411893									ao
s639415639				cca					
s649405895				cca					
s659406000				cca					
s668945602						cmc		ci	
s679407859				cca					
s689407842	aps					cmc		ci	
s699407834	aps					cmc		cnnn	
s709407180				cca					

Figure 7: Visualising the Solution, using Visual Basic: the exam timetable for each individual student

5. CONCLUSION

We have presented an open architecture and language for constraint programming, which greatly reduces the amount of knowledge required by the user to use Constraint Satisfaction technology, which is open and is thus easily extendible to use other constraint solvers. We have given an example showing how a real application can be simply and easily built, in the chosen language of the user, which uses our architecture for solving exam timetabling problems. This demonstrates that our proposed architecture is easy to use, open and extensible and is capable of solving real problems.

7. REFERENCES

[1] Corne, D., Fang, H-L., & Mellish, C., "Solving the Modular Exam Scheduling Problem with Genetic Algorithms", DAI Research Paper No. 622, Department of Artificial Intelligence, University of Edinburgh, UK.

[2] The CHIC-2 project, See http://www-icparc.doc.ic.ac.uk/chic2/.

[3] Freuder, E.C. & Mackworth, A., (ed.), "Constraint-based reasoning", MIT Press, 1994.

[4] Haralick, R.M. & Elliot, G.L., "Increasing Tree Search Efficiency for Constraint Satisfaction Problems", Artificial Intelligence 14 (1980) 263-313.

[5] Kunstmann, T. & Muller, R., "A Constraint Based Language for Spreadsheets", Proceedings of PACT96, pages 445-452, 24-26th April 1996, London, UK.

[6] Marriott, K. & Stuckey, P.J., “Programming with constraints, an introduction”, MIT Press, 1998.

[7] Meier, M. & Schimpf, J., “An Architecture for Prolog Extensions”, Proceedings of the 3rd International Workshop on Extensions of Logic Programming, Bologna, 1992.

[8] Michel, L. & Van Hentenryck, P., “Helios: A Modeling Language for Global Optimization”, Proceedings of PACT 96, pages 317-336, 24-26th April 1996, London, UK.

[9] Mills, P., Tsang, E., Williams, R, Borrett, J., Ford, J., “EaCL: An Easy abstract Constraint programming Language”, Technical Report CSM-321, Department of Computer Science, University of Essex, Colchester, UK, December 1998.

[10] Puget, J-F., “A C++ Implementation of CLP”, Proceedings of SPICIS 94, November 1994, Singapore.

[11] Puget, J-F., “PECOS: a high level constraint programming language”, Proceedings of SPICIS 92, September 1992, Singapore.

[12] Simonis, H., “The CHIP system and its applications”, in Montanari, U. & Rossi, F. (ed.), Proceedings, Principles and Practice of Constraint Programming (CP'95), Lecture Notes in Computer Science, Springer Verlag, Berlin, Heidelberg & New York, 1995, 643-646.

[13] Templeman, J., “Beginning MFC COM Programming”, Wrox Press, 1997.

[14] Tsang, E., “Foundations of Constraint Satisfaction”, Academic Press 1993.

[15] Van Hentenryck, P., Michel, L. & Deville, Y., “Numerica: A Modeling Language for lobal Optimization”, MIT Press, spring 1997.

[16] Van Hentenryck, P., “The OPL Optimization Programming Language”, MIT Press, 1999.

[17] Voudouris, C., “Guided Local Search for Combinatorial Optimization Problems”, PhD Thesis, Department of Computer Science, University of Essex, Colchester, UK, July, 1997.