

**An Overview Of The CACP Project:  
Modelling And Solving Constraint Satisfaction/Optimisation Problems  
With Minimal Expert Intervention**

**Richard Bradwell, John Ford, Patrick Mills, Edward Tsang, Richard Williams**  
Department of Computer Science  
University of Essex  
Colchester CO4 3SQ UK  
{[rbrad](mailto:rbrad@essex.ac.uk), [fordj](mailto:fordj@essex.ac.uk), [millph](mailto:millph@essex.ac.uk), [edward](mailto:edward@essex.ac.uk), [rjw](mailto:rjw@essex.ac.uk)@essex.ac.uk}

**Abstract**

*The constraint programming research community has accumulated a vast amount of experience in solving constraint satisfaction/optimisation problems. Unfortunately, applying constraint technology to a particular problem requires expertise in the technology, which many potential users do not have. The CACP project attempts to provide a system that encompasses the entire process of applying constraint technology. It supports the tasks of problem formulation and entry, in addition to supplying pre-written solvers and aiding the user in choosing which of the available algorithms to apply. Users may specify their problems using a declarative language. The problem specification is decoupled from the solvers, so the users may experiment with different problem formulations easily. Solvers supplied include a generalized Forward Checking solver, a Linear Programming solver and local search solvers implementing Guided Local Search, Tabu Search and Genetic Algorithms. A carefully designed interface is provided to guide users in understanding the technology.*

## Introduction

Within the Constraint Satisfaction/Optimisation research community a large amount of effort has been invested in engineering stronger algorithms, studying problem difficulty and more recently studying the implications of using different problem formulations [Tsang 1993; Freuder & Mackworth 1994]. As a result individual researchers, and the research community in general, have accumulated a large amount of implicit and explicit domain knowledge regarding how best to solve problems using constraint technology. With the knowledge required to apply constraint technology effectively, it is difficult to transfer the technology to an industrial setting without requiring an expert in the field.

Knowledge required to apply constraint technology effectively includes

- Knowing how to formulate a problem as a CSP/COP.
- Knowing how to engineer a solver.
- Knowing a good formulation for a given problem.
- Knowing which solver to apply to a given problem.
- Knowing how to incorporate domain knowledge into the solver.

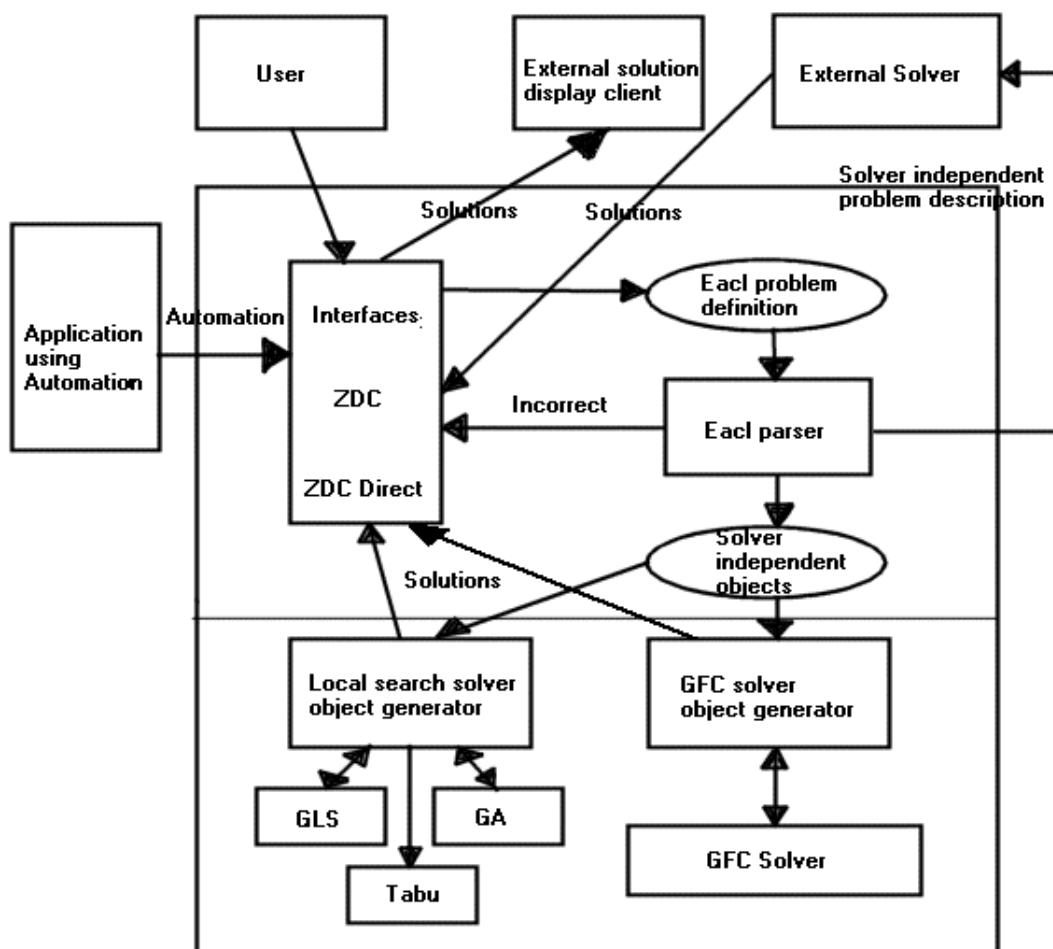
Various industrial strength packages, e.g. ILOG Solver, have been implemented with the explicit aim of making access to constraints technology easier (<http://www.ilog.fr>). Even packages such as these require a large amount of expertise because they usually come in the form of a constraint library that can be linked to a standard application, written in the desired 3GL. Knowledge of the target language and the constraint library is still required. Generally they concentrate only on the issue of solving, relieving the user from the burden of writing their own solver.

The CACP project attempts to provide a system that encompasses the entire process of applying constraint technology. It supports the tasks of problem formulation and entry, in addition to supplying pre-written solvers and aiding the user in choosing which of the available algorithms to apply. Problems are modelled in the declarative EaCL language, which the user can enter via an intuitive user interface. The problem specification is decoupled from the solvers, so the users may experiment with different problem formulations easily. Careful attention has been directed to providing a user friendly GUI based system for easy entry of problem constraints. An extensive help system is also provided. Having formulated the problem in the EaCL language and entered the problem, the user can solve the problem using one of the pre-written generic solvers. Choosing the correct solver from a problem is often a difficult task and therefore the CACP project makes an initial attempt to address this issue also.

Within the research community competition between algorithms drives researchers to produce faster algorithms that produce better results, for a restricted set of problems. Generally this is accomplished by tailoring the algorithm using large amounts of domain knowledge. The goal of the CACP project is not to compete with these highly specialised algorithms. In many industrial settings solutions are produced manually and a large amount of time and effort is invested in producing those solutions. Any solution to the problem, provided by a highly automated, easy to use system, produced within a reasonable time frame is usually what is desired. This is the premise on which the CACP project was built. The system implemented is primarily targeted for users who are not interested in implementing constraint programming techniques, but would like to exploit constraint technology.

## Architecture

The CACP architecture is given in Figure 1. The main flow of control starts with the entry of the problem definition in a language developed within the group, named EaCL. Additional data, required for more demanding problems, can be read in automatically from Excel by using it as an automation server. Problem description entry is performed using either the ZDC or ZDC Direct user interface. ZDC Direct allows the user to enter the problem formulation as text. ZDC uses a more elaborate interface to shield the user from the EaCL grammar, making problem entry easier.



**Figure 1: The CACP Architecture**

Once a problem description has been entered by the user, the EaCL parser parses the description. An invalid formulation is reported back to the user via the user interface. A correctly parsed definition generates the raw, solver independent constraint objects. These solver independent objects are used by the selected solver as a guide to generate its own set of constraint objects. This architecture essentially

separates the solver and its object library from the parser, allowing the architecture to be easily extensible. It is important to allow this separation because EaCL potentially supports a large number of different problem types. Each solver specialises in solving a particular class of problem and therefore requires a different set of constraint objects from solvers solving other problem types.

An algorithm selection expert module is responsible for matching the problem formulation to the available solver algorithms that can potentially be applied to that formulation. A solver, when invoked, runs in a separate thread, with only one thread executing at a time. The thread terminates once the solver has found a solution, or the solver has been terminated prematurely. The results from the solver are passed back to the user interface, where they are relayed to the user. Described above is the normal usage of the system. ZDC can also perform the role of an automation server. This means that standalone applications with specific user interfaces can be written for a particular domain, yet as automation clients they can access some of the problem solving capabilities of ZDC. The group, to demonstrate this feature, has developed a timetable planning application.

ZDC can also perform the role of a problem description server, using sockets. This means that ZDC can provide an external solver with a description of the variables, domains and constraints of the currently parsed problem. This facilitates a greater separation between the two separate phases of problem modelling and solving. The EaCL Parser generates a solver independent description of the problem. The description instructs the external client solver how to generate a constraint object tree, using its own constraint object library. The separation of modelling and solving is useful for a number of reasons. Firstly, it becomes very easy to add additional solvers to the system. External solver clients register with the server and the server becomes aware of their existence. An external solver can submit itself as a slave solver by sending the appropriate message to the server. As a slave, the external solver is under direct control of the server. The user can interact with the ZDC interface and force an external solver to solve the current problem being modelled and then return the results, as if it was an internal solver.

A number of external solvers, registered as slaves, can be instructed to solve the same problem at the same time. The use of the client/server model using sockets allows external solvers to be separate processes, possibly running on different machines over a network. Solutions, when received from the external solvers by the server, are stored for later viewing by the user. External solvers do not have to submit themselves as slave solvers. They can work autonomously and control their interaction with the server. In this manner the external solver can use ZDC as a problem description provider or solution displayer. GLS and its constraint object library have been externalised, to demonstrate the client/server architecture.

We intend to modify the server so that it can support different types of clients to be register with the server. It would, for example, be desirable to provide clients that display solutions in a manner appropriate to the problem being solved. The server will send solutions, when produced, to all registered display clients. Only those capable of displaying the solution will do so. To summarise, the CACP project, through its use of automation and sockets provides an open architecture. This allows applications to exploit the existing framework and tailor it more to a specific application domain, if required.

## **Problem Modelling in EaCL**

The Easy abstract Constraint Optimisation Programming Language (EaCL) is the language used to formulate problems prior to solving. A valid problem formulation is split into data, domains, variable, constraints and optimisation sub-sections. EaCL supports variables with integer, boolean, real and sets as domains. The EaCL grammar supports a wide range of logical, integer, set and symbolic constraints. There are also various facilities supporting lists and sets as well as conditional branching. For a complete description of the EaCL language consult [Mills et al 1998]. Modelling problems is one of the core activities and the main strength of the CACP project. For this reason we present an in-depth tutorial, demonstrating how to model a number of well-known test-bed problems in the Appendix.

## The Algorithm Expert

Providing a system that allows multiple solvers is strength in the sense that it allows a larger array of problem types to be solved by the system. Even having multiple solvers that are applicable to the same type of problem is advantageous because it allows greater scope for solving a given problem successfully. The weakness of maintaining a portfolio of algorithms is that it introduces additional choice to the user. Choosing the correct algorithm potentially requires a large amount of domain expertise. To simplify the choice it is suggested that an algorithm expert be used.

The algorithm expert is a function which, given a particular problem instance as its input, identifies from the portfolio of algorithms those which are applicable to the problem type. A secondary function of an algorithm expert is to take the set of algorithms applicable to the current problem class and reduce it further to the set it believed to perform best for the current problem instance. The first task in selecting a suitable algorithm is to reduce the set of algorithms based on the type of problem. A heuristic is incorporated in ZDC, which identifies Linear Programming (LP) problems and allows only the Simplex method to be applied to those problems. For other problem types such as problems containing variables with discrete domains, all of the incomplete heuristic based algorithms and the GFC algorithm, can potentially be applied to solve these problems.

Once a set of algorithms has been identified according to the problem type there is the possibility of further reducing the choice of algorithms. ZDC takes a pragmatic approach to algorithm selection by switching between algorithms mid-run if it is found that the current algorithm is ineffective. When switching takes place no solutions are passed between algorithms and each algorithm restarts from a new random solution. A single run involves trying all applicable algorithms in a given order until one of the algorithms succeeds in returning a valid solution, or the list of algorithms is exhausted. The algorithm expert attempts to predict based on previous runs, the probability of success for each algorithm and then orders them in the run accordingly to its predication. The solvers the algorithm expert believes to be the most promising are applied early in the run so there is a greater chance of early successful termination. This is partly based on the assumption that a user is likely to use ZDC to solve similar types of problems.

As a first attempt, the algorithm expert has been implemented to select its preferred algorithm ordering probabilistically, based on how well the algorithms have performed in the past. A simple weight is maintained for each algorithm. Roulette wheel selection is used to select the ordering of the algorithms using the algorithm weights. An algorithm's weight is used as its slot size in the selection process and algorithms that have previously performed well have a large slot size biasing the selection process towards them. A heuristic built into the expert ensures that GFC is always ordered first in the run, based on the observation that an exact search should take precedence over an incomplete search. Knowledge can be built into the individual solvers to identify when they are making insufficient progress and they can terminate before the end of their allotted "time slice". The forward checking algorithm [Haralick & Elliott 1980] has been implemented with a thrashing detecting mechanism developed in [Borrett et al 1996] for detecting if the search is thrashing. How to detect if an incomplete search algorithm is unlikely to make any further progress is still an open research area. Therefore, at present, we have no corresponding heuristic for the incomplete methods.

## The Solvers

A number of generic solvers have been implemented within the CACP framework. Firstly, a simplex algorithm is used to solve problems containing variables with real domains. A Generalised Forward Checking (GFC) algorithm has also been implemented, with a corresponding library of constraint objects, to represent the complete search algorithms. The forward checking algorithm, when applied to an optimisation problem, maintains the best solution cost and uses it as a bound on the current solution cost. GFC has also been extended so that it can be applied to problems involving n-ary constraints. As mentioned previously, built into the GFC algorithm is a thrashing detection mechanism that detects if the solver is an inefficient method for solving the current problem. Upon detection of thrashing the GFC is automatically terminated. Three local search techniques have been implemented, all sharing the same library of constraint objects. The solvers implemented are a Genetic Algorithm, Tabu Search and Guided Local Search. Initially we focus on GLS in detail because it demonstrates some of the principles used in the other algorithms.

### The Guided Local Search Solver

Guided Local search (GLS) [Voudouris & Tsang 1999] is a meta-heuristic. When the hill climber is caught in a local minimum it provides a mean of escaping the local minimum. Essentially GLS escapes from a local minimum by adding extra penalty terms to the cost function. When the algorithm detects it is in a local minimum it chooses a feature of the current solution to penalise. A term is then added to the cost function to increase the cost of any solution containing the penalised feature. Penalising the feature results in an increasing in the cost of the local minimum. Neighbouring solutions that do not exhibit the penalised features become more desirable, and hill climbing re-commences.

GLS has been implemented, in the project, to choose a downward move at random if one is available. A move is defined as unlabelling the chosen variable and re-labelling it with a different value. A downward move is one that reduces the number of conflicts the chosen variable is involved in. If no downward moves are available, then random sideways moves are tried. After a small number of consecutive sideways moves the escape mechanism is activated. The escape mechanism penalises features to escape local minima. In the context of constraint satisfaction, candidates for penalisation should be related to the current constraint violations. The variables involved in a constraint violation and their corresponding labels are used as features.

In an effort to speed up the GLS algorithm there are options to ignore constraints of arity greater than four. There is an additional option that allows only those constraints initially violated to be considered. The remaining constraints are only considered when a solution is found that does not violate those constraints initially violated.

### Label Inputs Table

A necessary step of local search is to examine the candidate neighbouring solutions. The GLS and Tabu Search (to be elaborated later) implementation maintains a table called the label inputs to make this task easier. The label inputs table holds, for each variable, the output of the constraints for each possible label in the variable domain. In effect the list indicates the number of conflicts a variable will be involved in if it is relabelled, for all domain values. The effects of the objective function are also added to the label inputs when solving optimisation problems. An example of how exactly the label inputs are calculated is given below.

Figure 2 formulates a simple problem in EaCL. The formulation describes a problem with two integer variables  $a$  and  $b$ . The product of the two variables is constrained to equal to a target value. The sum of  $a$  and  $b$  is minimised. If random labels are chosen for the variables then the label inputs can be calculated. For example, if  $a$  and  $b$  have the values of two and five respectively then the corresponding label inputs are

give in table 1. Each column in the table represents a variable. Each row in the table represents a variable labelling. For example, the contents of cell (b, 6) indicates the amount the constraint would be violated if b was given the label value six. We notice that this change would reduce the constraint input of variable b from its initial value of 1630 to 666. This is not surprising because labelling variable b with the value of six satisfies the constraint. The actual values given in the table cells need further explanation.

	Variable a	Variable b
Label 0	1506	1256
Label 1	1584	1339
Label 2	1630	1419
Label 3	1730	1497
Label 4	1836	1573
Label 5	1924	1630
Label 6	2009	666
Label 7	2095	1797
Label 8	2179	1906
Label 9	2262	1997
Label 10	2347	2086
Label 11	2430	2172
Label 12	2513	2256

**Table 1: An example of how label changes alter cost (current values in shade)**

```

Problem:test
{
    Data
    {
        target := 12 ;
    }
    Domains
    {
        IntDom d = {0,target} ;
    }
    Variables
    {
        IntVar a::d ;
        IntVar b::d ;
    }
    Constraints
    {
        a*b = target ;
    }
    Optimisation
    {
        minimise(a+b) ;
    }
}

```

**Figure 2: An Example Problem Formulation**

The values for the label inputs are calculated by adding the output produced by the constraint to the output produced by the expression being minimised. For example, the label input of variable b given the label five is calculated as follows. The left hand side (lhs) of the constraint is evaluated, resulting in a value of (2×5=) ten. The right hand side (rhs) of the constraint is twelve. The absolute difference between the lhs and the rhs (given the variable named dist) is used to calculate the output of the constraint as shown below. The value of m and energy are constants set at 10 and 1000 respectively.

$$ConOutput = (m + 1) * dist * energy / (m * dist) + 1$$

The output of the constraint given the values of a and b above, is (10+1) × 2 × 1000 / (10 × 2 + 1) = 1047. The output of the minimise expression is calculated by

$$MinOutput = coeff * energy * v / r$$

Both coeff and energy in the above equation are constants. Coeff is set to 2. The variable  $v$  is the value of the expression to minimise (which is  $a + b$ , in this case  $v = 2 + 5 = 7$ ). The variable  $r$  is the range of possible values the expression can take, in terms of the maximum minus the minimum value of  $a+b$ . Since both  $a$  and  $b$  may take values from 0 to 12,  $r$  is equal to  $(24 - 0 = ) 24$ . The output of minimise is therefore  $(2 \times 1000 \times 7 / 24 = ) 583$ . The resultant label input, calculated by summing ConOutput and MinOutput has a value  $(1047 + 583 = ) 1630$ .

## **The Tabu Search Solver**

The code developed for GLS, with those features described above, is a useful basis for any local search algorithm tackling COP's. Essentially the search strategy used by the algorithm can be separated from the details of maintaining information essential to local search such as the label inputs. Based on this observation a Tabu Search (TS) variant was developed to demonstrate that the GLS code could be re-used in this way.

The Tabu Search implemented in the project uses the label inputs to choose the most aggressive move at each step [Glover 1989, 1996]. A random variable is chosen for re-labelling at every iteration. The label inputs for that variable are examined and a list of the best possible non-tabu labels is constructed. The best are those that share the smallest value for their entry in the label inputs. Stated another way, those values that leave the chosen variable in the minimum number of conflicts are given greater priority. A new label for the current variable is chosen randomly from this minimum conflict list. If the chosen value does not represent a downhill move then, another variable is chosen randomly. When a non-tabu, downward move is found it is immediately taken. If all variables are examined and no non-tabu downward move exists, a random variable is chosen and is assigned a random value from its minimum conflict list. Before a variable is assigned a new value from its domain, the old label is stored on the tabu list. That old labelling of the variable remains tabu for a number of iterations. The exact number of iterations is determined by choosing a random number from a fixed interval.

## **The Genetic Algorithm Solver**

A Genetic Algorithm (GA) has also been implemented [Holland 1975; Goldberg 1989]. The GA evaluates each solution contained in the population and ranks them based on their fitness. The fitness of each individual is simply the sum of the outputs of all the constraints for that solution. Once ranked, the best N% of solutions is taken to form a breeding pool. From the breeding pool a new population is constructed by iteratively selecting two parents from the mating pool at random and using the crossover operator to produce an offspring solution. The offspring is placed in an empty slot in the new generation. This process continues until the new population is filled with solutions. The new population replaces the old population and the next round of breeding commences. The current implementation of the GA has found to be ineffective on the problems it has tested it on. We speculate the reason for this poor performance is that the GA finds it difficult to search a space consisting of a large number of different permutations all exhibiting the same cost.

## **Acknowledgements**

We wish to thank both Nathan Barnes and James Borrett for their contribution to the CACP project. This project is funded by the EPSRC grant GR/L20122.

## References

- Borrett, J.E., Tsang, E.P.K. & Walsh, N.R., Adaptive constraint satisfaction: the quickest first principle, Proceedings, 12th European Conference on AI, Budapest, Hungary, 1996, 160-164
- Freuder, E.C. & Mackworth, A., (ed.), Constraint-based reasoning, MIT Press, 1994
- Glover, F., Tabu search Part I, Operations Research Society of America (ORSA) Journal on Computing 1, 1989, 109-206
- Glover, F., TABU search and adaptive memory programming -- advances, applications and challenges, in Barr, Helgason & Kennington, (ed.), Interfaces in Computer Science and Operations Research, Kluwer Academic Publishers, 1996
- Goldberg, D.E., Genetic algorithms in search, optimization, and machine learning, Reading, MA, Addison-Wesley Pub. Co., Inc., 1989
- Haralick, R.M. & Elliott, G.L., Increasing tree search efficiency for constraint satisfaction problems, Artificial Intelligence, Vol.14, 1980, 263-313
- Holland, J.H., Adaptation in natural and artificial systems, University of Michigan press, Ann Arbor, MI, 1975
- Mills, P., Tsang, E.P.K., Williams, R., Ford, J. & Borrett, J., EaCL 1.0: an easy abstract constraint programming language, Technical Report CSM-321, University of Essex, Colchester, UK, December, 1998
- Tsang, E.P.K., Foundations of constraint satisfaction, Academic Press, London and San Diego, 1993
- Tsang, E.P.K., Mills, P., Williams, R., Ford, J. & Borrett, J., A computer aided constraint programming system, The First International Conference on The Practical Application of Constraint Technologies and Logic Programming (PACLP), London, April 1999, 81-93
- Voudouris, C. & Tsang, E.P.K., Guided Local Search and its application to the Travelling Salesman Problem, European Journal of Operational Research, Anbar Publishing, Vol 113, Issue 2, March 1999, 469-499

## Appendix – Example Problem Formulations

### Case study 1: The Travelling Salesman Problem(TSP)

#### Problem Definition

A salesman must visit a number of cities on business and then return back to his start city. Each city is visited only once and all cities must be visited. The objective is to minimize the overall distance traveled

#### Problem Formulation

An EaCL formulation for the TSP is given below;

```
1      Problem:TSP
2      {
3          Data
4          {
5              distances := [0,60,57,120,200,
6                          60,0,115,175,210,
7                          57,115,0,0,100,260,
8                          120,175,100,0,120,
9                          200,210,260,120,0] ;
10             Max := 5 ;
11         }
12         Domains
13         {
14             IntDom Num = (0,Max-1) ;
15         }
16         Variables
17         {
18             IntVar Cities[Max]::Num ;
19         }
20         Constraints
21         {
22             AllDifferent(Cities) ;
23         }
24         Optimisation
25         {
26             Minimise
27             {
28                 Sum( [ distances[(cities(x)*Max)+cities[x+1]] | x in [0 .. Max-2]
29                     ++ [ distances[(cities[Max-1]*Max)+cities[0]] ] )
30             } ;
31     }
```

Lines 3-11 in the problem definition declares the data used in the problem. The constant 'Max' defines the number of cities to visit in the problem. The list 'distances', declared in lines 5-9, defines the geometric distances between each city. The domain section, lines 12-15, defines a single integer domain 'Num' containing values between 0 and 'Max'-1. A TSP solution is represented by the array of constrained variables 'Cities', declared in line 18. Each position in the 'Cities' array will contain a city number,

therefore the array has a domain of 'Num' associated with it. Position 'n' in the array will hold the n'th city visited in the tour. To ensure no city is visited more than once the AllDifferent constraint is used. In line 22 the AllDifferent constraint is applied to the array 'Cities'. This constrains each variable to have a different city value (or each position to have a different city number). Lines 24-31 declare the problem optimization sub-section of the problem definition. The entire expression, given in line 28, minimizes the distance of the current tour. The sub-expression

**[(distances[(cities(x)\*Max)+cities[x+1]] | x in [0 ..Max-2])**

generates a list of distances between each successive city held in the array 'Cities'. The distances between the cities is "looked up" in the list of distances, using the city numbers as an index. This list is concatenated, using the concatenation operator ++, to a list containing the cost of the edge between the first and last cities. The final edge in the tour is given by the sub-expression

**[ distances[(cities[Max-1]\*Max)+cities[0]] ]**

The 'Sum' function summates all the values held in the entire list to give a final tour cost.

[End of case 1]

## Case study 2: The Car Sequencing Problem

### Problem Definition

The car sequencing (CS) problem concerns manufacturing a number of cars on a production line. Every car is based on a prototype car with additional features added. Additional features are added to the cars at dedicated stations. There is a single station for each possible additional feature. Stations have been designed such that only a given number of cars in a sequence of a certain length can be supplied with the feature. The problem is to find a sequence of cars that does not violate the capacity constraints of the stations while ensuring that every car has the correct options installed.

### Problem Formulation

An EaCL fomulation for the Car Sequencing problem is given below;

```
1      Problem:CarSequence
2      {
3          Data
4          {
5              Options_Capacities := [[1,2],[2,3],[1,3],[2,5],[1,5]];
6              Cars_Required := [[0,1,1,0,1,1,0],
7                               [1,1,0,0,0,1,0],
8                               [2,2,0,1,0,0,1],
9                               [3,2,0,1,0,1,0],
10                              [4,2,1,0,1,0,0],
11                              [5,2,1,1,0,0,0]];
12          }
13          Domains
14          {
15              IntDom D=[0, #Cars_Required-1];
16          }
17          Variables
18          {
19              IntVar Cars[Sum([Cars_Required[i][1] | i in [0..#Cars_Required- 1]])::D;
20          }
21          Constraints
22          {
23              //Requirements constraint
24              Sequence(Cars, [Cars_Required[i][0] |
25                          i in [0..#Cars_Required-1], j in [1..Cars_Required[i][1]]]);
26              //Capacity constraints
27              Constraint LimitCapacity(Cars, Option, Max, OutOfMax)
28              {
29                  //Forall lists x, x is a subset of Cars of length OutOfMax
30                  Forall (subseqs seq of Cars, #seq = OutOfMax)
31                  {
32                      Count(seq, [class | class in [0..#Cars_Required-1],
33                               Cars_Required[class][option+2] = 1]) <= Max;
34                  }
35              }
36              Forall (j in [0..#Options_Capacities-1])
37              {
38                  LimitCapacity(Cars, j, Options_Capacities[j][0], Options_Capacities[j][1]);
39              }
40          }
41      }
```

Lines 5-11 in the problem formulation declare the problem data. The list 'Options\_Capacities' holds sub-lists that correspond to station capacities. For example the second sub-list (2,3) indicates that station one can only fit the feature it is responsible for, to any two out of a given sequence of three cars. The list 'Cars\_Required' holds the details of the cars to manufacture. Each sub-list corresponds to a particular class of car, each fitted with different options. The first value in a sub-list indicates the class number of the car being manufactured. The second value indicates the number of cars of the current class to manufacture. The remaining values in the sub-list are Boolean values indicating if each corresponding feature should be included for this type of car.

Line 19 declares the array of constrained variables 'Cars', the assignment of values to which, will give a solution to the problem. The sub-expression

**[ Cars\_Required[i][1] | i in [0..#Cars\_Required-1] ]**

iterates through each car class entry in turn, extracts the number of cars of that class required and adds it to the temporary list. The # operator gives the length of a list. The Sum function then sums all the numbers held in the list. This gives the total number of cars required and a corresponding length for the declaration of the 'Cars' array.

Lines 21-38 declare the problem constraints. There are two types of constraints, the requirement constraints and the capacity constraints. The requirement constraints ensure that the correct number of cars, for each class of car, is manufactured. Line 24 enforces the requirement constraints using the sequence constraint. The sequence constraint ensures that each variable in the array, given as its first argument, is assigned a value from the list given as its second argument. Each value in the list can only be assigned once. In this example the sequence constraint ensures that each member of 'Cars' is assigned the value of a car class. The sub-expression

**(Cars\_Required[i][0] | i in [0..#Cars\_Required-1], j in [1..Cars\_Required[i][1]])**

utilises two loops to generate the correct number of cars in the list, for each car class.

Lines 34-37 declare the capacity constraints on the individual stations. Each sub-list, representing an individual station within the 'Option\_Capacities' list is used to create a corresponding 'LimitCapacity' constraint. The 'LimitCapacity' constraint is a user-defined constraint, declared at lines 26-32. The sub-expression on line 29 is very important

**Subseqs seq of Cars, #seq = OutOfMax**

because it creates a subset of the current car orderings. The size of the subset depends on the capacity constraint being generated. For example if the station could only provide every two out of three cars with the features it offers, then only sequences of length three need be considered. The Forall keyword ensures that all sequences of the correct length are examined. The count keyword, introduced in line 31, counts the number of occurrences of the current feature in the subsequence being examined. The less than or equal to constraint ensures that the count is less than the number of cars that can be accommodated by the station.

[End of case 2]