

*School of Computer Science and Software Engineering  
Monash University*

Bachelor of Computer Science Honours  
Clayton Campus

Literature Review  
2002

*Compiling OPL to HAL*

(Virginia) Lee Mei Leng 12738018

Supervisors: Dr Maria Garcia de la Banda, A/Prof Kim Marriott

Second Reader: Prof David Abramson

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Constraint Solving and Constraint Optimization</b>	<b>4</b>
<b>3</b>	<b>Solver Techniques</b>	<b>5</b>
3.1	Mathematical Programming . . . . .	5
3.1.1	Integer Programming . . . . .	7
3.1.2	Mixed Integer Programming . . . . .	7
3.2	Constraint programming . . . . .	7
3.3	Meta-heuristic . . . . .	8
3.3.1	Local Search . . . . .	8
3.3.2	Simulated Annealing . . . . .	9
3.3.3	Tabu Search . . . . .	10
<b>4</b>	<b>Constraint Solving and Constraint Optimization Tools</b>	<b>10</b>
4.1	Spreadsheet . . . . .	10
4.2	Custom application written in traditional programming language . . . . .	11
4.3	Custom application written in higher-level programming language . . . . .	11
4.4	Constraint Logic Programming . . . . .	12
<b>5</b>	<b>Modelling Language</b>	<b>12</b>
5.1	Problem Modelling . . . . .	14
5.2	Search Modelling . . . . .	14
5.3	Optimization Programming Language . . . . .	15
<b>6</b>	<b>Discussion</b>	<b>16</b>

# 1 Introduction

Combinatorial problems are a special class of problems involving decision variables and constraints. Constraints dictate the architecture and fundamental properties of the problem. More importantly, constraints describe the expected outcome of the decision variables, by specifying their domain. Typical combinatorial problems are timetabling, staff scheduling, resource allocation, planning and the travelling salesman problem (TSP). Most combinatorial problems are classified under the NP-hard problem category and, therefore, there is no known algorithm to solve them in polynomial time. As the dimension of the problem increases, the time needed to solve the problem grows exponentially. Unfortunately, large organisations encounter these problems on a day-to-day basis, and the lack of an efficient algorithm affects the effective utilization of their resources. Since resources often constitute a big part of total cost in large organisations, efficient solving of staff scheduling for instance can end up saving thousands and even millions of dollar.

The need to solve combinatorial problems effectively has attracted much attention from the research community and remains as one of the most actively researched areas. Consequently, the research community has devoted a lot of effort to address this issue computationally. This has resulted in three main approaches of solving combinatorial problems, namely Operation Research (OR) also widely known as Mathematical Programming(Winston 1995), Constraint Programming and meta-heuristic(Reeves 1993) (normally associated with Artificial Intelligence (AI)).

Mathematical programming uses equation solving techniques to solve combinatorial problems. In contrast to Mathematical Programming, constraint programming focuses more on applying a judicious search strategy that exploits the problem structure to enumerate possible solution in an efficient manner. Meta-heuristic is an ad-hoc approach in which the search algorithm is based on a collection of heuristics techniques (or “rules of thumb”). These heuristics techniques range from probabilistic and memory-based to actual simulation of natural processes.

All three fields have developed substantially over the years and obtained variable degrees of success. Generally, if a combinatorial problem needs an optimal solution, then mathematical programming techniques such as Simplex would seem a good fit provided the problem could be modelled as linear equations. However, if only a feasible solution is required, the problem is a good candidate for applying constraint programming techniques. The application of meta-heuristic techniques in combinatorial solving are not well understood. Due to this uncertainty, meta-heuristic techniques often served as an alternative approach when the other two techniques could not produce desired solutions.

Even though all three approaches have orthogonal strengths and weaknesses, researchers generally agreed that hybrid algorithms generate better results compare to conventional approaches (Hooker 2000). Although solving techniques play a big part in combinatorial problem solving, there are other issues involved to ensure a combinatorial problem is solved efficiently. One such issue is modelling. A problem that is represented inaccurately can affect the solving efficiency. Modelling languages are tools that facilitate the task of modelling. In particular, they allow the user to apply solving techniques without understanding the underlying working mechanism.

In this paper, attention is paid to literatures that cover different aspects of a modelling language and different approaches towards modelling. In doing so, we hope to gain some insights to

assist in the research of adding modelling capabilities onto HAL, a constraint logic programming language.

In Section 2, we defined the formal definition of constraint solving and constraint optimization. In Section 3, we discuss solver techniques arise from mathematical programming, constraint programming and meta-heuristic. In Section 4, different tools that facilitate the application of solving techniques are discussed. Finally, in section 5, modelling languages are discussed.

## 2 Constraint Solving and Constraint Optimization

The techniques associated with solving combinatorial problems can generally be categorized into two groups, namely constraint solving and constraint optimization (Meyer 2000). Constraint solving is defined as obtaining a solution within the constraint domain (which specifies the range of acceptable values) that satisfies all constraints. Usually, than one solution satisfies such conditions. Constraint solving is normally associated with constraint satisfaction problems, a class of combinatorial problems that require all constraints to be satisfied to obtain a solution. Sometimes, the focus of a constraint satisfaction problem is not about the solution, but rather, whether the problem is solvable or not. Therefore, satisfying all constraints is not an option but a necessary condition in a constraint satisfaction problem. Constraints under this definition are also known as hard constraints. An example of the constraint satisfaction problems is the map colouring problem with only three colours available. More precisely, under mathematical programming terminology a constraint satisfaction problem is defined as follows:

Given  $n$  domains  $D_1, D_2, \dots, D_n$  and  $m$  constraints  $f_1, f_2, \dots, f_m$ , find  $n$  variables  $x_1, x_2, \dots, x_n$  such that

$$\begin{aligned} f_k(x_1, x_2, \dots, x_n) &= 1, & 1 \leq k \leq m \\ x_j &\in D_j & 1 \leq j \leq n \end{aligned}$$

where each constraint is represented as a function  $f$  with  $n$  variables. These  $n$  variables are restricted to take values within their domain, as specified by  $x_j \in D_j$ . When a constraint  $k$  is satisfied, the function  $f$  will return a value of 1. This is analogous to a conjunction of  $m$  boolean variables, where violation of any of the constraints result in a false outcome.

Constraint optimization, although similar to constraint solving, is comparatively harder because it only accepts a solution provided the optimality of a given objective function is proven. Thus, it has to deal not only with a list of hard constraints but also with an objective function. The objective function is a criterion that determines how well a solution fulfills the objective. Obviously, the objective function is a soft constraint, because it is desirable but not mandatory to satisfy the objective function.

Normally, optimality is proved by comparing current solution with the best solution found so far. Therefore, for any constraint optimization problem the optimal solution is the best possible solution, and one that best fulfills the objective function. An example of the constraint optimization problems is the travelling salesman problem (TSP) (Reinelt 1994) with minimizing travelling distance as the objective function. This is a constraint optimization problem because

only the solution with the shortest travelling distance is produced, provided the problem is solvable. Finally, the definition of a constraint optimization problem under the mathematical programming terminology is as follows:

$$\begin{aligned}
 & \text{Minimize} && g(x_1, x_2, \dots, x_n) \\
 & \text{Subject to} && f_k(x_1, x_2, \dots, x_n) \quad 1 \leq k \leq m \\
 & && x_j \in D_j \quad 1 \leq j \leq n
 \end{aligned} \tag{1}$$

There are some similarities between the above formalization and an earlier formalization for a constraint satisfaction problem. Indeed, both formulas share the same hard constraints. The only difference is the above formalization includes an objective function.

### 3 Solver Techniques

A solver is an algorithm that implements a constraint solving or constraint optimization technique. (Marriott & Stuckey 1998) Normally the technique involved is very simple, the main idea of a solver is to repeatedly apply the same solving process until the problem is solved. A solver will terminate the iterative process if all possible solutions are exhausted. In other words, a solver automates the process of solving.

Just like there are different classes of combinatorial problems, there are numerous techniques for constraint solving and constraint optimization. The best technique to use depends on the problem. Several techniques are considered in this section. First, techniques from mathematical programming's approach are considered. This is followed by a discussion about techniques from constraint programming. Finally, we will consider alternative techniques from meta-heuristic.

#### 3.1 Mathematical Programming

Mathematical programming has its roots in Operations Research back in the 1940s and is used to solve problems such as planning, scheduling, maximizing profits and minimizing costs (Lustig & Puget 2001). Mathematical programming requires all constraints associated to the problem to be represented mathematically as a set of equations or inequalities involving variables. A problem modelled in such format can be solved using for example, Gauss-Jordan elimination. The success of mathematical programming is largely due to the ability to solve linear problems efficiently. All linear problems are described in terms of an objective function and linear inequalities (simply known as constraints). The objective function is a linear function describing the problem to be maximized or minimized, and it is governed by a conjunction of linear inequalities.

Arguably, the Simplex method is by far the most popular method for solving linear problems. The method is explained in the following scenario, a leather factory manufactures two types of leather belts, the deluxe model and the normal model.

Each type of belt requires 1 sq yd of leather. Manufacturing a deluxe belt requires 2 hour of skilled labour, while a normal belt only requires 1 hour of skilled labour. Each week, only 40

sq yd of leather and 60 hours of skilled labour is available. A deluxe belt has a profit of \$4 and a normal belt has a profit of \$3. The company wants to maximize total revenue.

To begin, the problem is formulated as follow:

$$\begin{aligned}
 x_1 &= \text{number of deluxe belts produced} \\
 x_2 &= \text{number of normal belts produced} \\
 \\ 
 \text{maximize } & z = 4x_1 + 3x_2 \\
 \text{subject to } & x_1 + x_2 \leq 40 \quad (\text{Leather constraint}) \\
 & 2x_1 + x_2 \leq 60 \quad (\text{Labour constraint}) \\
 & x_1, x_2 \geq 0
 \end{aligned}$$

This problem can be modelled graphically as in the following Figure 1.

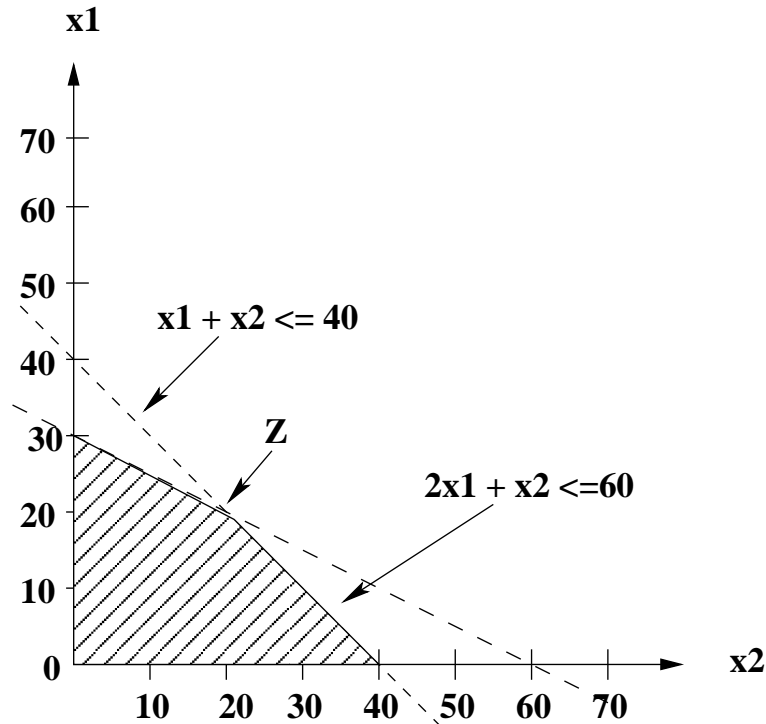


Figure 1: Simplex solving method

In figure 1 the feasible region is shaded, and the two constraints are represented as linear functions. Both functions intersect at point Z. There are 4 extreme points in the feasible region, those points are known to be feasible solution. Each of these extreme points correspond to the intersection of the linear functions defining each constraints. Because the optimal solution lies in one of these points, the simplex algorithm inspects each feasible solution in turn until the optimal solution is discovered. In this case, the solution is point Z with location (20, 20), thus the maximum profit obtainable is \$140 with 20 belts of each type produced weekly.

In the worst-case scenario, the simplex algorithm has an exponential execution time. However, in practice, the algorithm is well behaved and solves most problems in reasonable time. In

linear programming, it is implicit that all variables take on real values. Problems that require variables to take on integer value fall under another category of mathematical programming called integer programming.

### **3.1.1 Integer Programming**

Strictly speaking, integer programming is a refinement of linear programming, and results in problems that are usually harder to solve. However, integer programming models some real world problem more realistically. For example, in problems such as rostering, scheduling employees cannot take on real values. Integer programs are normally solved using the branch-and-bound method, which starts by solving the linear programming relaxation of the integer programming. If the optimal solution assumes integer values, it is also the optimal solution to the integer problem and we have finished. On the other hand, if the optimal values obtained are of real values then the process will branch out to create two sub problems based on the floor and ceiling of the real values. Then, one of the sub problems is chosen, the constraints are reinstated, and choices that are sub-optimal are discarded. The method finds the optimal solution by efficiently enumerating the points in the reduced feasible region. The branching process terminates when all variables obtained integer values and further branching is not possible. All solutions obtained from sub problems are then compared and the best solution found so far is the optimal solution.

The main limitation of integer programming is that the solving process becomes very inefficient as the number of variables increases. Thus, for problems with large number of integer variables, other techniques such as Column Generation are used. Column Generation increases the size of problems that can be solved while limiting the solving time to a reasonable time frame.

### **3.1.2 Mixed Integer Programming**

Mixed integer programming is a hybrid of integer programming and linear programming in which some variables are required to be integers while the others can take real values.

Solving mixed integer problems generally involves using the same branch-and-bound method with a slight modification. First, the linear relaxation of the problem is solved to obtain a partial solution. Based on this partial solution, further branching only occurs on variables that are required to be integers. Similar to integer programming, every new branching instruction creates two new sub problems. Each sub problem will generate solution or leads to further branching. By repeating the same process, this method will exhaust all feasible solution and generate a solution ultimately.

## **3.2 Constraint programming**

Constraint programming consists of three main components, namely the constraints, the constraint solving mechanism(also known as the solver) and the search procedure. The constraints describe the problem architecture and any associated fundamental properties. The solver implement the algorithm of a solving technique. The search procedure essentially specifies how to find solutions without violating the constraints. Constraint programming has been successfully applied to many combinatorial problems, in particular, planning and scheduling. For these

kind of problems, the constraint solving mechanism has usually been based on two fundamental techniques: constraint propagation and domain reduction, which work as follows. First, the solver will examine the domain of all variables, removing values that are inconsistent with the constraints placed on the variable. Then, the domain of the variables is updated to reflect the changes. A variable with an empty domain indicates there is no solution for the problem, and earlier steps can be undone. Domain updates propagate changes to constraints that interact with the variable. This in turn results in propagation of domain reductions to other variables. At the end of the day, the domain of variables is reduced to a manageable size. These two techniques compliment each other and together they determine if a problem is satisfiable. The order in which propagation and reduction are applied to the variables, is determined by the search strategy. This can be a default search strategy provided by the constraint programming system or a custom-made search strategy defined by the user. Generally, the default search strategy provided is based on backtracking style, depth first search. This term comes from the analogy that the strategy traverses the search tree downwards(depth first) until there is no node to visit in that direction. If that happens, then the search will backtrack to the parent node and select another child node to proceed. The whole searching mechanism is explained as follows. In the search tree, the root node contains all the initial values of the variables. At each step, a variable is chosen and every value in the domain of the variable creates a new leaf node. All these leaf nodes are called choice points, because each leaf node corresponds to a specific choice. Once the creation of leaf nodes for a variable is complete, a node is chosen and the value is propagated to other variables. A variable with an empty domain indicates there is no feasible solution, and the system automatically backtracks to the last choice point (parent node) and tries other leaves of the same parent. The search will continue until a solution is found or until no feasible solution is located after exploring the entire search tree. Search strategy combined with domain reduction and constraint propagation effectively reduced the search space.

### **3.3 Meta-heuristic**

Meta-heuristic is a collection of heuristic algorithms applicable to a wide variety of problems. The heuristic approach is often associated with “rules of thumb” or clever insights. Based on the heuristic provided, the algorithm performs the search process iteratively to look for a solution. The iterative search process is terminated when no improvements are possible. The choice of meta-heuristic algorithm depends on a few factors, such as the solution quality required and the availability of problem knowledge. Because meta-heuristic is a rather ad hoc technique, there is no guarantee a solution can be obtained. In most cases, meta-heuristic techniques served as the last resort when the other two techniques could not produce the desired solutions. In the following section, several popular meta-heuristic techniques are discussed.

#### **3.3.1 Local Search**

Local search is an iterative transition process that starts with an initial solution and produces a sequence of solutions by applying small local changes. If a new solution rates better when compared with the best current solution, it replaces the current best solution. The whole searching process halts upon reaching the maximum number of iterations, or when no improvement seems possible. The solution to the problem is the best solution found so far. Local search works well

when the problem is linear, because in those cases, the local optimum is also the global optimum. However, in non-linear cases where there are several local optimums, a bad starting point can cause local search to terminate upon encountering the first local optimum. This problem is illustrated in the following diagram:

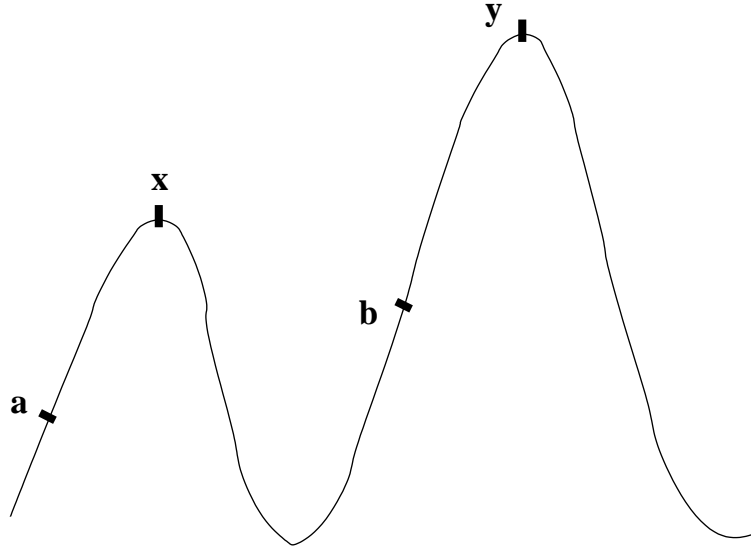


Figure 2: Local optimum problem

In Figure 2, the global optimum  $y$  is the solution. If the local search algorithm starts at point  $a$ , then the solution  $x$  is returned because it is the local optimum. However, if the algorithm were to start from  $b$  then we are guaranteed to get  $y$  returned as the solution. Thus, a good starting point plays an important role in local search, but to overcome local optimality one has to adopt other approaches like simulated annealing and tabu search.

### 3.3.2 Simulated Annealing

Simulated annealing (Russell & Norvig 1995) is a probabilistic search algorithm. The term simulated annealing derives from the process of heating and then cooling a substance slowly to finally arrive at the solid state. The search algorithm simply mimics the physical process as follows.

In the early stages of the execution, the temperature is high resulting in higher probability, and cause jumping to occur more frequently. Jumping occurs as a way of avoiding local minima, accepting a poorly performed solution with a higher probability. Otherwise, any solution performing better than the current solution is accepted and replaced as the best solution found so far. As the execution time elapses, the temperature decreases thus inadvertently reduces the frequency of jumping. This probabilistic nature of the system guarantees the exploration of other solution space instead of terminating whenever encountering the first local optimum.

The simulation process terminates after a number of successive executions with no improvements, and returns the best solution so far. The only drawback of simulated annealing is the long execution time to obtain quality solutions. Although it is possible to achieve global optimum

solution, it comes at a cost of slower cooling procedure and longer iteration at each temperature level. Conversely, if in favour of shorter execution time, the algorithm compromises the solution quality.

### 3.3.3 Tabu Search

Tabu search (Glover & Laguna 1997) is a search algorithm that employs memory-based strategies. The principal idea behind tabu search is an intelligent system, which takes previous experiences into account before deciding the current move. This approach contrasts with most memoryless systems that rely heavily on probability in the form of sampling.

Tabu search begins with an initial solution marching towards local minima. As mentioned earlier, the memory system forms the basics of tabu list: a list of previous moves collected during the search. The tabu list disallows any moves that occur in the list. By keeping the inverse of last  $n$  moves in a tabu list, this method also eliminates the possibility of looping. Unlike simulated annealing that accepts a poor solution provided it has high probability, tabu search will accept a poor solution to avoid reinvestigating a path. If the algorithm cannot identify any solution, it then chooses the best neighbour. This again allows the algorithm to avoid local minima and ultimately finding the desired solution.

## 4 Constraint Solving and Constraint Optimization Tools

Constraint Solving and Constraint Optimization Tools are tools that facilitate the application of constraint solving and constraint optimization techniques. Such tools can be categorized into four categories: spreadsheet programs serving as interface to a solver, custom applications written in traditional programming languages (like Fortran and Cobol), custom applications written in modern programming languages (like Visual Basic and C++) and constraint logic programming. In the following section, all four approaches are analysed and discussed.

### 4.1 Spreadsheet

Most spreadsheet programs like Microsoft Excel come with a set of powerful inbuilt mathematical functions and calculation facilities. There are several advantages when using one of such systems to model problems. First, they minimise the learning curve for a user who is already familiar with such systems. Second, most tasks are automated and the user can use charts and graphs to display the results in a more intuitive format. However, when the model becomes larger, it becomes difficult to specify. This is especially true when hundreds or more formulas are involved. Errors become increasingly difficult to track down and other programmers waste much effort trying to understand the model. Apart from that, there are other issues involved such as extendibility. Because most solvers are provided in spreadsheet program as plug-ins, the user has no access to the workings of the solver. An example of a popular plug-in solver for spreadsheet is What's Best. The broad applicability of spreadsheet programs is also one their strengths and weaknesses. Due to this, spreadsheet programs have limited flexibility in modelling. Any subsequent changes to the original model are hard to modify without introducing new errors. Such situations make it harder to add new dimensions to the model.

Advantages	Disadvantages
A complete set of library functions is available at disposal	Harder to manage as model gets larger
Ability to display results in charts and graph	Difficult to add new dimensions to model
Translation from model to solver understandable format is automated	

## 4.2 Custom application written in traditional programming language

Creation of a custom application is not easy, because the user has to possess some programming skills. Also, the user has to play two different roles, both as a modeller who translates the problem into a model and the programmer who programs the solver. To make things harder, any changes to the model requires modifying the program. The task of managing the program becomes harder when the problem gets more complex. Even though users have to endure all these hardships, in return, they are rewarded with faster execution time and greater control over the execution of the program. With greater controllability, the user can experiment with hybrid solving methods

Advantages	Disadvantages
User has full control over the execution of the program	Harder to manage program as the problem gets more complex
Faster execution time	Model changes requires rewriting the program
Allows experimentation of hybrid methods	Programming skills required
	Program needs to translate model into solver understandable format

## 4.3 Custom application written in higher-level programming language

Most custom applications are written in higher-level programming languages because they provide better modelling support than traditional programming languages. For example, constraints can be modelled as objects. This type of custom applications provide constraint solving abilities otherwise not available in higher-level programming languages. Some common examples are Pecos for Lisp (*Pecos: A High Level Constraint Programming Language* 1992), Screamer for Common Lisp (Siskind & McAllester 1991), (Siskind & McAllester 1993), ILOG Solver for C++ (team 1999), CPLEX (team 2000) for C++ and Modeller++ for C++ (Michel & Hentenryck 2001b). Strictly speaking, these custom applications can be seen as merely library extensions to higher-level programming language. From this point of view, custom applications also support different solvers in higher-level programming language simply by extending the libraries. Because the solvers are created before hand and can be applied to a wide variety of problems, user are not required to know the working details of a solver.

Even though users are granted access to a variety of solvers through library interfaces, this has limited the extendability of the solvers. Furthermore, accessing solvers through library

interfaces implies the user has limited control over the execution of a program. With limited control ability, users are restricted from creating or experimenting with hybrid solvers.

Advantages	Disadvantages
Allows the development of search techniques	Limited extendability
Allows experimentation with different solvers	Limited control over the execution of the program
User do not need to understand the working mechanism of solvers	User has no access to create or experiment hybrid solvers

#### 4.4 Constraint Logic Programming

Constraint logic programming provides constraint solving and constraint optimization techniques with user-defined constraints and customized solver interface. Constraint logic programming uses a declarative style syntax. which is easy to use and understand. As a programming language, users are given the ability to program the search strategy and even to extend the underlying solver. The support for user-defined constraints allows problems to be modelled easily. However, to use the system users are required to posses some constraint programming skills. Generally, this type of language has a slower execution time compared to traditional programming language like C.

Advantages	Disadvantages
Shorter development time	Constraint programming skills required
More control over the execution of the program	Slower execution time
Customized seach technique	
User extendibility solver giving greater flexibility	
Easy to use declarative style syntax	

## 5 Modelling Language

Modelling Languages are languages with a generic expressive syntax that allow problems to be expressed in the most intuitive and general form. The main aim of a modelling language is to facilitate the process of modelling. More precisely, the idea is to allow the user to create a model without understanding the details of the underlying solving mechanism. Modelling languages achieve this by providing a front-end generic syntax to which the user are accustomed. Any problem modelled in this syntax automatically gets translated to a solver understandable format. Because most solvers' format are very similar, the translation enables problems modelled in the modelling language to access a wide variety of solvers. In fact, most modelling languages provide a list of state of the art solvers to experiment with the problem model. Also, due to the generic nature of modelling language, a wide variety of problems can be modelled easily and efficiently.

Another feature that distinguishes modelling languages from dedicated programming languages is the ability to separate model and data instance. This separation enables the user to reuse the same model for different instance of problems without modification.

Modelling languages are as expressive as general programming languages because any expression expressed in a general programming language can also be expressed in a modelling language. In addition, modelling languages have a slight advantage over general programming languages, because the user is not required to possess any programming skills. Due to the high-level abstraction of the language, problems modelled in modelling languages generally have fewer lines of code. The reduction in the number of lines of code has a significant impact when it comes to maintainability.

Also, modelling languages conceal the underlying cumbersome code to piece together a language that allows problems to be modelled more realistically and intuitively. Such usability is not directly achievable in general programming languages, instead, the programmer has to assign meaningful names to each variable associated with real world objects. From there on, the programmer is responsible for maintaining the consistent use of variable names throughout the entire program. To illustrate on this idea, let us consider the following example. Here, the idea is to model a list of routes between two destinations. The code represented here is written in OPL, the C code is left as an exercise for the interested reader.

```
enum location { bendigo, clayton, geelong, melbourne, ballarat, sunshine };
struct route { location a; location b; };
{route} connection = { <bendigo, clayton>, <clayton, melbourne>,
                      <melbourne, sunshine>, <sunshine, geelong>,
                      <ballarat, bendigo> };
int distance[connection] = [ 45, 20, 12, 32, 17 ];
```

Notice in the above example, how a set variable connection is declared to exploit the sparsity in the problem. Such facility is not available in traditional programming languages. Instead, the user has to assign a number to each location and create a (5 x 5) two dimensional array to store the value. Out of the 25 spaces allocated only 5 values are stored in the array. Obviously, this is a waste of space.

Just like there are different techniques for solving combinatorial problems, there are different areas of modelling. The two main areas of modelling are problem modelling and search modelling. Problem modelling focuses on representing the problem more accurately and thus making it easier to understand. The model should be understandable by both the modeller and the solver. Essentially, the model describes what the problem is without mentioning how to solve it. AMPL(Fourer, Gay & Kernighan 1998) and GAMS(Bisschop & Meeraus 1982) are two examples of modelling languages that focuses on problem modelling.

The focus of search modelling is slightly different compared to problem modelling. Search modelling aims at simplifying the task of specifying a search technique. The complexity is partly due to the lack of support from traditional programming languages. In these languages, a user who wishes to specify a search technique is implicitly required to have significant programming skills. Ideally, the model should describe the search technique in a format understood by both the modeller and the underlying system. The second part is especially important because it ensures the search technique is translated into proper algorithmic form and applicable to any problems. For search modelling, there are modelling languages like Localizer and SALSA. However, among all these modelling languages Optimization Programming Language (OPL) stands out as a special case, because OPL supports both problem modelling and search modelling.(Hentenryck,

Michel, Perron & Régis 1999), (van Hentenryck, Perron & Puget 2000), (Hentenryck, Michel, Laborie, Nuijten & Rogerie 1999) In following sections, problem modelling and search modelling will be discussed in order. Then, it is followed by another discussion on OPL as a hybrid modelling system.

## 5.1 Problem Modelling

In mathematical modelling languages, a problem is represented as a list of equations with each unknown entity appearing as a variable. As a mathematical modelling language, AMPL(Fourer et al. 1998) models problem as equations and inequalities. Expressions in AMPL are therefore expressed using traditional algebraic notations. Such expressions provide excellent support for modelling a special class of combinatorial problems like maximizing profit, because naturally this is how these problems are described in the first place. Another pleasing feature of AMPL is the separation of model and data. This feature enables the same model to be applied to other instance of problem without modifying. The only drawback of AMPL is the awkwardness associated with modelling a certain class of problems. A good example is the map colouring problem. In AMPL the model uses to represent countries and colours. Obviously, it is an awkward and unnatural way to model the map colouring problem.

Such modelling problem is overcome by introducing higher-level.

ESRA(Pierre Flener 2001), (Hnich 2001), (Flener 2001) is a modelling language created based on OPL. It is both a subset and a superset of OPL, because some of ESRA's syntax are higher-level and more expressive than OPL, while the rest is just identical to those of OPL. ESRA only offers problem modelling ability, this makes ESRA a dedicated problem modelling language. ESRA's limited ability has forced the language to translate all the expressions to OPL code to be useful. The expressiveness of ESRA is obtained by supporting more advanced data structures like sequence, bags and sets. When translating these data structures to OPL, the data structures are simply rephrased as another lower level data structure with several imposed constraints. Strictly speaking, ESRA merely functions as a translator, translating from one syntax to another.

## 5.2 Search Modelling

A search algorithm constitutes a big component of both constraint solving and constraint optimization. A good search algorithm can effectively cut down the possible solution space, and locate the solution in an efficient manner. Take the Simplex method for example, the algorithm applies strategic search techniques that explore the solution space from one basic feasible solution to another. This strategy is based on the insight that the optimal solution lies in one of the basic solutions in the feasible region (solution space). Having a search modelling language enables the user to implement and tailor the search technique for a specific problem model. In the rest of this section, a few modelling languages customized for search algorithms are discussed. The discussion will include local search oriented modelling languages like Localizer and a spin off of an even more generalized search language SALSA.

As a typical modelling language, Localizer(Michel & Hentenryck 2001a) aims at making the specification and implementation of local search algorithms easier. In Localizer, there are three concepts that control the workings of a search procedure, namely Invariants, Neighbourhood

and Control. Invariants are statements like constraints that maintain variable changes during the course of execution. Neighbourhood is a selection criterion function that determines the neighbouring states at any time during the execution. As for Control, it is an acceptance criterion determining whether a solution is accepted. Implementing choice is akin to implementing several acceptance statements with an associated probability. This is also analogous to Tabu list in the Tabu search and the probability of temperature versus degradation of quality in Simulated Annealing. The most important concept Localizer has initiated is the possibility of providing a generic syntax for customizing the implementation of local search.

In the same vein as Localizer, SALSA (Laburthe & Caseau 1998) was produced for modelling search algorithms. The idea of treating control and logic separately has greatly influenced the creation of SALSA. As a result, SALSA is created as a dedicated search modelling language. In other words, SALSA is not a standalone language but rather works cooperatively with another host language. In fact, all SALSA expressions are translated to CLAIRE, a constraint programming language. Compared to Localizer, SALSA is even more generic, because it offers the ability to specify local, global and even hybrid search algorithms. The specification of a hybrid search algorithm is a complicated one; this is due to the need to keep track of possible disjoint variables used by both techniques during the search. In SALSA, this difficulty is overcome by creating a generic syntax that caters for both the similarities and differences of both local search and global search. Thus, this generic syntax enables different search techniques to be specified easily.

### 5.3 Optimization Programming Language

OPL (Hentenryck 1999), (Hentenryck, Michel, Perron & Régim 1999), (van Hentenryck et al. 2000), (Hentenryck, Michel, Laborie, Nuijten & Rogerie 1999) is the first modelling language that combines constraint programming and mathematical programming techniques for solving combinatorial problems. The design of OPL was motivated by mathematical modelling languages like AMPL, and the success of constraint programming techniques in solving combinatorial problems. The main difference between OPL and other modelling languages is that OPL provides support for both modelling and search aspects of constraint programming. In other words, OPL allows the user to specify not only the problem but also the search procedure tailored to the problem. The ability of OPL to unify two different techniques, simplifies the experimentation of hybrid techniques in solving combinatorial problems. However, OPL is incomplete in terms of programming language. This is because OPL does not have the ability to program constraint solving algorithms. One of the novelty concepts introduced by OPL is activity, a special class of data type dedicated solely to solve scheduling related problems. Other facilities provided by OPL are higher order constraints, logical combinations of constraints and support for enumerated types, sets, range, structures, variables and array indexed by arbitrary variables. A typical example of higher order constraints is as follow  $sum(jinRange)(s[i] = i)$ , here every time the relation  $s[i] = i$  evaluates to true, the total value of summation will increase by one. This elegant statement captures one of the essential constraints in the magic series problem (Kraitichik 1942).

Following is a table that summarizes the advantages and disadvantages of OPL. From the table, it is not hard to notice Consider the following OPL program:

Advantages	Disadvantages
Generic syntax that is close to natural language or mathematical notation	User has limited control over the execution of the program
Higher-level syntax represents the problem more concisely	Slower execution time
State-of-the art solvers available straight out from the toolbox	
Programming skills not required	
Model reuse for different instance of data	
Support both mathematical programming and constraint programming	

## 6 Discussion

Clearly, a powerful hybrid modelling language like OPL has simplifies combinatorial problem solving. However, when comparing OPL to a constraint logic programming language like HAL, OPL failed to cover all possible constraint domains. On the other hand, HAL lacked the expressiveness and modelling ability of OPL. It is the need to have the abilities of both languages that has initiated the idea of extending HAL. This enables HAL to handle mathematical programming and ease the task of modelling. However, one important question remains, which is how to make such extension feasible. The answer to this question is obtained through the course of investigating relevant literatures in this area. Based on literatures that are covered in this paper, there are two ways where modelling functionalities can be supported in HAL. The first method is to extend the language library to cover the desired modelling functionalities. Such implementation approach was adopted by Localizer++ and Modeller++. The second approach was inspired by ESRA and SALSA, it is rather naive, basically a compiler is created to translate the model from one language to another. However, neither method alone is sufficient because both methods compliment each other. Based on these insights, we have come to the conclusion to adopt the following methods in the research. First, a compiler will be created to translate OPL syntax to HAL. At the same time, HAL's library will be extended to support mathematical functionalities as available in OPL.

## References

- Bisschop, J. & Meeraus, A. (1982). *On the Development of a General Algebraic Modeling System in a Strategic Planning Environment*, Mathematical Programming Study.
- Flener, P. (2001). Towards relational modelling of combinatorial optimisation problems.
- Fourer, R., Gay, D. M. & Kernighan, B. W. (1998). *AMPL: A Modeling Language for Mathematical Programming*, The Scientific Press, San Francisco, CA.
- Glover, F. & Laguna, M. (1997). *Tabu search*, Kluwer Academic Publishers, Boston.
- Hentenryck, P. V. (1999). *The OPL Optimization Programming Language*, The MIT Press, Cambridge, Massachusetts.
- Hentenryck, P. V., Michel, L., Laborie, P., Nuijten, W. & Rogerie, J. (1999). Combinatorial optimization in OPL studio, *Portuguese Conference on Artificial Intelligence*, pp. 1–15.
- Hentenryck, P. V., Michel, L., Perron, L. & Régim, J.-C. (1999). Constraint programming in opl, in G. Nadathur (ed.), *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, Vol. 1702 of *Lecture Notes in Computer Science*, pp. 98–116.  
\*citeseer.nj.nec.com/hentenryck99constraint.html
- Hnich, B. (2001). High-level modelling and reformulation of constraint satisfaction problems, *Lecture Notes in Computer Science* **2239**: 766–??
- Hooker, J. N. (2000). Logic, optimization, and constraint programming, *Technical report*, Graduate School of Industrial Administration, Carnegie Mellon University.
- Kraitchik, M. (1942). Magic series 7.13.3, *Mathematical Recreations* pp. 143 and 183–186.
- Laburthe, F. & Caseau, Y. (1998). SALSA: A language for search algorithms, *Lecture Notes in Computer Science* **1520**: 310–??
- Lustig, I. J. & Puget, J.-F. (2001). Program does not equal program: Constraint programming and its relationship to mathematical programming, *Interfaces Vol 31 No 6*: 29–53.
- Marriott, K. & Stuckey, P. J. (1998). *Programming with Constraint: An Introduction*, The MIT Press, Cambridge, Massachusetts.
- Meyer, B. (2000). Cse 460 optimisation and constraint solving.
- Michel, L. & Hentenryck, P. V. (2001a). Localizer++: An open library for local search, *Technical Report CS-01-02*, Brown University, Computer Science Department.
- Michel, L. & Hentenryck, P. V. (2001b). Modeler++ a modeling layer for constraint programming libraries, *Technical report cs-00-07*, Brown University.
- Pecos: A High Level Constraint Programming Language* (1992). Proceedings of First Singapore International Conference on Intelligent System(SPICIS), Singapore.
- Pierre Flener, B. H. (2001). The syntax and semantics of esra, *Astra internal report*, Computing Science Department, Uppsala University, Box 256, 751 05 Uppsala, Sweden.

- Reeves, C. (1993). *Modern Heuristic Techniques for Combinatorial Problems*, Halsted Press (Wiley).
- Reinelt, G. (1994). *The Travelling Salesman: Computational Solutions for TSP Applications*, Springer-Verlag Lecture Notes in Computer Science 840.
- Russell, S. J. & Norvig, P. (1995). *Artificial intelligence : a modern approach*, Prentice Hall, Englewood Cliffs, N.J.
- Siskind, J. M. & McAllester, D. A. (1991). Screamer: A portable efficient implementation of nondeterministic common lisp, *Technical Report IRCS-93-03*, Philadelphia, PA.
- Siskind, J. M. & McAllester, D. A. (1993). Nondeterministic lisp as a substrate for constraint logic programming, in R. Fikes & W. Lehnert (eds), *Proceedings of the Eleventh National Conference on Artificial Intelligence*, AAAI Press, Menlo Park, California.
- team, I. (1999). Ilog solver 4.4 users manual, *Technical report*, ILOG, Gentilly, France.
- team, I. (2000). Ilog cplex 7.0 reference manual, *Technical report*, ILOG, Gentilly, France.
- van Hentenryck, P., Perron, L. & Puget, J.-F. (2000). Search and strategies in OPL, *ACM Transactions on Computational Logic* 1(2): 285–320.
- Winston, W. L. (1995). *Introduction to Mathematical Programming*, Belmont, Ca. : Duxbury Press.