A New Approach to Integrating Mixed Integer Programming and Constraint Logic Programming

Robert Rodošek, Mark G. Wallace and Mozafar T. Hajian

IC-Parc, Imperial College
London SW7 2AZ, England
E-mail: {rr5, mgw, mh10}@doc.ic.ac.uk

This paper represents an integration of Mixed Integer Programming (MIP) and Constraint Logic Programming (CLP) which, like MIP, tightens bounds rather than adding constraints during search. The integrated system combines components of the CLP system ECLiPSe [7] and the MIP system CPLEX [5], in which constraints can be handled by either one or both components.

Our approach is introduced in three stages. Firstly we present an automatic transformation which maps CLP programs onto such CLP programs that any disjunction is eliminated in favour of auxiliary binary variables. Secondly we present improvements of this mapping by using a committed choice operator and translations of pre-defined non-linear constraints. Thirdly we introduce a new hybrid algorithm which reduces the solution space of the problem progressively by calling finite domain propagation of ECLiPSe as well as dual simplex of CPLEX. The advantages of this integration are illustrated by solving efficiently difficult optimisation problems like the Hoist Scheduling Problem [23] and the Progressive Party Problem [27].

Keywords: Constraint Logic Programming, Mixed Integer Programming.

1 Introduction

Many applications can be naturally defined using a logical formalism, in a way similar to functional or logic programming. In declarative programming, or any runnable specification language, a *default evaluation algorithm* turns the logical formalism into a running program [4]. There are many drawbacks of a default algorithm, e.g, poor performance on many (indeed most) programs and frequent failure to terminate at all.

In constraint programming, by contrast, the behaviour of the constraints is equally as important as their definition. Constraints have two features, definition and behaviour, and these can be handled separately. The practical consequence is that the programmer can concentrate on modelling of the problem and the problems with performance and termination can be ironed out afterwards. Unfortunately, any disjunction in the logic program, expressed by multi-clause predicates, leads to the enumeration of different alternatives [10]. The evaluation algorithm, which is based on the implicit enumeration, is complete but not always efficient.

The contribution of this paper is represented in two parts. In the first part we present an efficient evaluation algorithm to turn the logical formalism with constraints into a running program on a set of only linear equalities and inequalities. In the second part we present an integration of the MIP solver and the CLP solver.

The proposed evaluation algorithm allows an integration of MIP with CLP using a unique model for a problem. Disjunctions appearing in a CLP program are mapped to 0/1 auxiliary variables. The program is translated into a generic MIP model which can be instantiated to a specific model as soon as the unknown data are supplied. The evaluation algorithm has an important property that avoids any implicit enumeration.

The derived conjunction of linear equalities and inequalities can be hence treated either by the MIP solver, the CLP solver or by both solvers. In CLP, many different local consistency algorithms can be used. They reduce domains of variables or detect an infeasibility if the domain of a variable becomes empty. The algorithms represent an efficient *local constraint propagation* [9]. For instance, constraint propagation on finite domains of variables is successful in solving problems arising in planning and scheduling, resource allocation, and more recently even in solving more complex numerical problems [22, 28]. In MIP, a simplex algorithm is used to solve the continuous relaxation of the problem, giving either an infeasibility, or a lower bound on the objective function. Simplex represents an efficient *global constraint propagation* [32]. MIP is a technique that has been applied to a wide range of complex optimisation problems [31]. Although a lot of combinatorial problems can be modelled in CLP as well as in MIP, the performances of the techniques may be very different [27].

We present an integration of the MIP solver with the CLP solver giving more powerful constraint reasoning: *local and global constraint propagation*. The integrated solver combines our CLP solver, ECLiPSe, with a commercial MIP solver, CPLEX. However, we confine our use of the commercial solver to the linear constraints, only using the dual simplex solver of the CPLEX. In effect we return to a system quite similar to CLP, but use an external solver for the linear rational constraints instead of an internal one. This is possible within our framework because the translation has enabled us to post all the linear constraints at the root of the search tree: the search is now reduced to the labelling (or alternatively domain splitting) of integer variables. Therefore, unlike CLP(R) [17], we do not require a linear constraint solver that can handle the incremental posting of constraints. The empirical results show that this combination offers genuine practical advantages over both the MIP and CLP solver. For example, we use a single program both to find and prove the optimality of the solution to the Progressive Party Problem [27].

The rest of this paper is organised as follows. Section 2 presents the handling of conjunction and disjunction within a possibly recursive CLP program. Section 3 presents a translation of a given CLP program to a generic model. Section 4 integrates the MIP solver with the CLP solver. Experimental results are demonstrated in Section 5. Finally, Section 6 concludes the paper.

2 Modelling in CLP

Many experimental systems concern logic and 0-1 inequalities [14, 31]. A systematic procedure for transforming a set of logical statements or logical conditions into an equivalent mathematical formulation has been presented by Williams [29, 30], and McKinnon and Williams [16]. Recently, a computer support for this task within a Mathematical Programming modelling system was given by Hadjiconstantinou et. al. [12]. The syntax of a Mathematical Programming language is extended to incorporate propositional logic terms with linear algebraic forms. A transformation of the operators like *implies, or, and, atmost k, exactly k* into a conjunction of linear constraints over 0-1 variables is described in [12]. The method is regarded as a bottom-up approach since the constraints have to be expressed using predefined operators.

In CLP the constraints are not restricted on predefined operators. CLP is more powerful as a representation language than Mathematical Programming, since it allows the representation of constraints in a more natural and compact way [26]. The problem can be represented by high level pre-defined and user-defined constraints. The modelling represents a top-down approach since the constraints are defined with further constraints. This property of constraints allows an additional insight to the structure of the problem [6]. For instance, the modelling can be used to recognise that there are many equally optimal solutions which can be cut off at an early stage of the CLP solution process.

In this section we present an efficient evaluation algorithm to turn the logical formalism with constraints into a running MIP program on a set of linear inequalities. The evaluation algorithm is performed in two steps. The first step represents an automatic translation of a given CLP program to a generic version of it. The syntax of CLP programs is defined in Subsection 2.1 while the translation directives are represented in Subsection 2.2. The second step carries out the unfolding algorithm described in Subsection 2.3 to derive a MIP model.

2.1 Conjunction and Disjunction Within a Recursive CLP Program

An *atom* is denoted by $p(t_1, ..., t_n)$ where p is an n-ary predicate symbol and t_1, ..., t_n are terms where each variable appears only once in any term. We distinguish between two kinds of atoms: "user atoms" and "constraints". The user atoms are defined by the programmer while the constraints are linear or type constraints. A *linear constraint* $X \leq Y$ represents a linear inequality where X and Y are linear expressions [1]. The conjunction of linear constraints $X \leq Y$ and $Y \leq X$ is denoted by $X = Y$ to represent a linear equality. The type of variables is defined by a *type constraint* using 1-ary predicate symbols *binary*, *integer* and *real* to represent binary, integer and real variables, respectively.

A *program P* is defined by a set of clauses in which each clause has a head and a body. The notation for a *clause* is

$$\mathtt{p(Args) :- q_1(Args_1),\ ...,\ q_k(Args_k).}$$

where the head is a user atom $p(Args)$ and the body is the conjunction of atoms $q_1(Args_1)$, ..., $q_k(Args_k)$.

[1]An expression $A_1 * V_1 + ... + A_k * V_k$ is called a *linear expression* iff A_i is a constant and V_i is a variable (or a constant) for every i, $1 \leq i \leq k$.

Example 2.1
A program to check whether the difference between two numbers chosen from 1 to 10 is greater than or equal to 2 can be defined in the CLP language as follows:

$$\begin{array}{lll}
\mathtt{prog(X,Y)} & :- & \mathtt{integer(X)},\ 1 \leq \mathtt{X},\ \mathtt{X} \leq 10, \\
 & & \mathtt{integer(Y)},\ 1 \leq \mathtt{Y},\ \mathtt{Y} \leq 10, \\
 & & \mathtt{diff(X,Y)}. \\
\mathtt{diff(X,Y)} & :- & \mathtt{Y} + 2 \leq \mathtt{X}. \\
\mathtt{diff(X,Y)} & :- & \mathtt{X} + 2 \leq \mathtt{Y}.
\end{array}$$

A program is executed by "unfolding" its clauses in response to a query. For instance, query `?-prog(X, Y)` causes the definition of `prog(X,Y)` to be unfolded into constraints `integer(X)`, $1 \leq \mathtt{X}$, $\mathtt{X} \leq 10$, `integer(Y)`, $1 \leq \mathtt{Y}$, $\mathtt{Y} \leq 10$, and, in turn, the definition of `diff(X,Y)` to be unfolded into constraint ($\mathtt{Y}+2 \leq \mathtt{X}$ `or` $\mathtt{X}+2 \leq \mathtt{Y}$). The result of the unfolding is a set of constraints. However, the unfolding of `diff(X,Y)` introduces a disjunction which cannot be directly handled by MIP. □

2.2 Translation to a generic model

The contribution of our translation is a redefinition of each atom $p(Args)$ which is defined by more than one clause (see atom *diff(X,Y)* in Example 2.1). Since any disjunction in the program is represented by two or more clauses with the same head, we translate such clauses into clauses with unique heads. The translation is based on the idea of adding an auxiliary binary variable B as a new argument to atom $p(Args)$. We say that the translated atom $p(Args, B)$ is equivalent to $p(Args)$ if $B = 1$. If $B = 0$, then $p(Args, B)$ is always true.

A *generic program*, $gen(P)$ is a program which is translated from P using translation directives: (i) on the goal in the language of P; (ii) on the clauses of the program; (iii) on the linear constraints; and (iv) on the type constraints in the body of those clauses:

Directive 1: *If an atom $p(Args)$ is a goal in the language of P, then the atom $p(Args, 1)$ is introduced:*

$$\mathtt{p(Args) :- p(Args,1).}$$

Directive 2: *If an atom $p(Args)$ is defined by r clauses*

$$\begin{array}{l}
\mathtt{p(Args_1) :- q_{11}(Args_{11}),\ ...,\ q_{1k}(Args_{1k}).} \\
\ldots \\
\mathtt{p(Args_r) :- q_{r1}(Args_{r1}),\ ...,\ q_{rk}(Args_{rk}).}
\end{array}$$

then we introduce r auxiliary binary variables and for every n-ary predicate symbol of the clause an $(n+1)$-ary predicate symbol. The clauses are then translated to the following clauses,

$$\mathtt{p(Args,B) :- p_1(Args,B_1),\ ...,\ p_r(Args,B_r),\ B_1 + ... + B_r = B.}$$

$$\mathtt{p_1(Args_1, B_1) :- q_{11}(Args_{11}, B_1), ..., q_{1k}(Args_{1k}, B_1), binary(B_1).}$$

...

$$\mathtt{p_r(Args_r, B_r) :- q_{r1}(Args_{r1}, B_r), ..., q_{rk}(Args_{rk}, B_r), binary(B_r).}$$

If an atom in the body of a clause is a linear constraint, then it has to be translated by adding auxiliary binary variable B into another linear constraint which is equivalent to the original constraint if B is instantiated by 1, and it is true if B is instantiated by 0.

Directive 3: *A linear constraint* $\mathtt{X} \leq \mathtt{Y}$ *is translated into a linear constraint*

$$\mathtt{X + M * B \ \leq \ Y + M}$$

where B is an auxiliary binary variable and M is a large enough constant, e.g. $M = max\{(upper(X) - lower(Y)), (upper(Y) - lower(X))\}$ where $upper(X)$ represents the upper bound of X and $lower(Y)$ represents the lower bound of Y.

Directive 4: *Type constraints are translated into the same type constraints.*

We show that the proposed directives preserve the set of all solutions of a given program. A *solution* of a program with goal $p(Args)$ is defined by an assignment of values to the variables of $Args$ returned by the usual evaluation of CLP programs [9].

Lemma 2.2
If $p(Args)$ is a goal in the language of a program P, then it has the same set of solutions when evaluated against P and against $gen(P)$.

Proof
To satisfy program P, called by $p(Args)$, one of the clauses with head $\mathtt{p(Args)}$ has to be satisfied. Let r be the number of such clauses. Using Directive 1, atom $p(Args, 1)$ has to be satisfied. If $r = 1$, then Directive 2 implies that atoms $q_{11}(Args_{11}, 1)$, ..., $q_{1k}(Args_{1k}, 1)$ have to be satisfied. If $r > 1$, then Directive 3 derives equality $B_1 + ... + B_r = 1$ which has to be satisfied. Without loss of generality, let $B_1 = 1$. It follows that atoms $q_{11}(Args_{11}, 1)$, ..., $q_{1k}(Args_{1k}, 1)$ of the clause with head $p_1(Args, 1)$ have to be satisfied. At the end of the translation there are only atoms of the form $q'(Args', 1)$ to satisfy. Since every atom $q'(Args', 1)$ has the same set of solutions as the original atom $q(Args)$ of P, goal $p(Args)$ has the same set of solutions when evaluated against P and against $gen(P)$. □

Since each clause of a program is translated only once, the complexity of the translation is $O(m)$ time where m is the number of clauses in the program.

2.3 Unfolding to an MIP model

When the input data of a given program are supplied, the generic program is automatically unfolded into a conjunction of linear and type constraints. An important consequence for the execution of practical applications is that the unfolding algorithm proceeds without backtracking.

Lemma 2.3
For a given input data of a program with goal p(Args) *the generic program is unfolded into a conjunction of constraints only:*

$$\mathtt{p(Args)} :- \mathtt{constraint}_1, \ldots, \mathtt{constraint}_s.$$

Proof
By using the directives, every user atom of the generic program occurs in the head of one clause. Furthermore, every constraint is a conjunction of other user atoms and/or constraints. When the input data has been supplied, the generic program is unfolded into a conjunction of constraints only. □

The proposed translation to a generic CLP program and the unfolding to an MIP model is illustrated using the problem in Example 2.1.

Example 2.4
A program to check whether the difference between two numbers is greater than or equal to 2 has been defined in Example 2.1. The program is first translated to the generic program:

$$\begin{aligned}
\mathtt{prog(X,Y)} \quad &:- \quad \mathtt{prog(X,Y,1)}.\\
\mathtt{prog(X,Y,B)} \quad &:- \quad \mathtt{integer(X)},\ \mathtt{B} \le \mathtt{X},\ \mathtt{X}+\mathtt{B} \le 11,\\
&\qquad \mathtt{integer(Y)},\ \mathtt{B} \le \mathtt{Y},\ \mathtt{Y}+\mathtt{B} \le 11,\\
&\qquad \mathtt{diff(X,Y,B)}.\\
\mathtt{diff(X,Y,B)} \quad &:- \quad \mathtt{diff}_1(\mathtt{X},\mathtt{Y},\mathtt{B}_1),\ \mathtt{diff}_2(\mathtt{X},\mathtt{Y},\mathtt{B}_2),\ \mathtt{B}_1+\mathtt{B}_2=\mathtt{B}.\\
\mathtt{diff}_1(\mathtt{X},\mathtt{Y},\mathtt{B}_1) \quad &:- \quad \mathtt{Y}+2+11*\mathtt{B}_1 \ \le\ \mathtt{X}+11.\\
\mathtt{diff}_2(\mathtt{X},\mathtt{Y},\mathtt{B}_2) \quad &:- \quad \mathtt{X}+2+11*\mathtt{B}_2 \ \le\ \mathtt{Y}+11.
\end{aligned}$$

The generic program is now unfolded into the following conjunction of constraints:

$$\begin{aligned}
\mathtt{prog(X,Y)} \quad :- \quad &\mathtt{integer(X)},\ \mathtt{B} \le \mathtt{X},\ \mathtt{X}+\mathtt{B} \le 11,\\
&\mathtt{integer(Y)},\ \mathtt{B} \le \mathtt{Y},\ \mathtt{Y}+\mathtt{B} \le 11,\\
&\mathtt{Y}+11*\mathtt{B}_1 \ \le\ \mathtt{X}+9,\ \ \mathtt{X}+11*\mathtt{B}_2 \ \le\ \mathtt{Y}+9,\ \ \mathtt{B}_1+\mathtt{B}_2=1.
\end{aligned}$$

□

3 Improving the Model

The proposed automatic translation in Section 2 does not necessarily achieve the most computationally efficient model. We present several improvements of our translation. First, using data/control-flow properties of programs [21], the number of the introduced auxiliary 0/1 variables can be reduced. Second, the proposed translation can also be extended to pre-defined constraints. Third, it is not necessarily to translate the whole program, since the user can decide which predicates should be considered by the CLP solver and which predicates by the MIP solver.

3.1 Data/control-flow properties of programs

In many programs the arguments of predicates are lists of constants and variables. The clauses usually satisfy data/control-flow properties [21] which lead to a simplification of the proposed translation.

If a problem is specified in detail, so that each predicate definition consists of an immediately determinable division into cases and each case breaks down the problem represented by the predicate into slightly smaller sub-problems, then execution can proceed with little or no need for backtracking. With this programming style it is possible to write programs which are both algorithmic and declarative.

Maher [21] presents a class of languages ALPS and some results which formalise these observations. Instead of clauses, a program is a finite set of *rules* of the form

$$H \Leftrightarrow G \mid B,$$

where H is a user atom (the head), and G (the *guard*) and B (the body) are conjunctions of atoms. The symbol $\mid$ is called the *commit operator.* The operational semantics of the commit operator is that a constraint H can commit to a given rule if the guard of just one rule with head H can be satisfied. Thus, the program of rules prevents alternative choices along the execution. If the head H can be satisfied with one rule, then it represents a "don't care" choice, otherwise it represents "don't know" choices since atom H can be satisfied by more than one clause.

Directive 5: *A rule R is translated to R.*

The consequence of this directive is such that the auxiliary binary variables are only generated for the clauses but not for the rules.

Let us demonstrate the proposed directives on a simple program where the goal is to guarantee a difference greater than or equal to 2 between a variable X and each element of a list L:

```
test(X,L)  ⇔   L = [] | true.
test(X,L)  ⇔   L = [Y|R] | diff(X,Y), test(X,R).
diff(X,Y)  :-  X + 2 ≤ Y.
diff(X,Y)  :-  Y + 2 ≤ X.
```

The program is translated using Directive 5 for `test`-atoms and Directive 2 for `diff`-atoms. The number of auxiliary variables for a given problem becomes smaller since the auxiliary variables are not introduced for the rules. If the length of the list L is k, then the translation of the committed-choice program generates only k auxiliary variables instead of $2k$ when using the program without rules.

Another example is a recursive program which defines a linear constraint between every two variables of a list containing k variables. If the program has no guards, then the translation introduces $k(k+1)/2$ auxiliary variables. By adding appropriate guards into the program, the number of auxiliary variables introduced by the translation can be reduced to 0.

3.2 Replacement of pre-defined constraints

In the previous section the constraints were restricted to linear and type constraints. The definition of programs is usually extended to allow other pre-defined constraints besides the linear and type constraints. For such constraints the user defines a translation to linear constraints. The restriction on integer variables to take only two values (0 or 1), is a property of most MIP models [27, 31]. The practical consequence of such models is usually better performance of the MIP solver [32].

We consider three pre-defined constraints:

1. $V\texttt{::}Lo\texttt{..}Hi$, which presents the finite domain $\{Lo, ..., Hi\}$ for an integer variable V.
2. $\texttt{link}(B, V_1, V_2)$, which presents the equivalence that variables V_1 and V_2 are equal if and only if binary variable B is equal to one.
3. $\texttt{alldistinct}(Vars)$, which does not allow the same value for any two variables of set $Vars$.

The language of programs is extended by a predicate symbol `replace`. This predicate symbol denotes that a pre-defined constraint is replaced by type and linear constraints as follows:

Replacement 1: *A constraint* $\texttt{replace}(V\texttt{::}Lo\texttt{..}Hi)$ *is replaced by the conjunction of type and linear constraints*

$$binary(B_{Lo}),\ binary(B_{Lo+1}), ...,\ binary(B_{Hi}),$$
$$B_{Lo} + ... + B_{Hi} = 1$$

where every binary variable B_i represents value i for variable V.

Replacement 2: *A constraint* $\texttt{replace(link}(B, V_1, V_2))$ *is replaced by type and linear constraints*

$$binary(B'),\ binary(B''),\ B + B' + B'' = 1,$$
$$V_1 + 1 + M * B' \leq V_2 + M,\ V_1 + M * B \leq V_2 + M,$$
$$V_2 + 1 + M * B'' \leq V_1 + M,\ V_2 + M * B \leq V_1 + M,$$

where M is a sufficiently large number w.r.t. the bounds of variables V_1 and V_2.

Replacement 3: *A constraint* $\texttt{replace(alldistinct}(\{V_1, ..., V_n\}))$ *is replaced by the conjunction of linear constraints*

$$B_{11} + ... + B_{n1} \ \leq\ 1,$$
$$...$$
$$B_{1k} + ... + B_{nk} \ \leq\ 1,$$

where every binary variable B_{ij} represents the assignment of value j to integer variable V_i, and k is the size of the domain of the variables.

Consider the last replacement rule, `alldistinct` constraint in a CLP program is replaced by the conjunction of linear constraints over binary variables. If the binary variables are defined in the program, then it is enough to apply the last rule Replacement 3. Otherwise, we have to apply all three replacement rules.

The replacement rules enable us to handle different constraints of a model by different solver. For instance, `alldistinct` constraint is usually handled by the CLP solver while the generated linear constraints are handled by the MIP solver. Two kinds of problems appear by such replacements.

First, during the unfolding process we need a global access to each unique binary variable B_{ij} representing value j for variable V_i. The maximum number of such binary variables is the product of the number of integer variables by the size of the largest integer domain. Note, the domain of variable V_i is recognised by constraints $d_1 \leq V_i$ and $V_i \leq d_2$ where d_1 and d_2 are constants. Therefore, the size of the domain for variable V_i is determined by $d_2 - d_1$.

Second, the replacement directives require us to define link constraints between integer and new binary variables like $\texttt{link}(B_{ij}, V_i, j)$. Unfortunately, an automatic translation of these constraints to linear constraints does not lead to an efficient mathematical model for simplex. To overcome these difficulties we use two strategies. The first strategy replaces the original constraints with linear constraints over binary variables and handles the `link` constraints, for example, by the CLP solver where they can "easily" be handled by the FD propagation. The second strategy simply ignores the `link` constraints and uses the global propagation just to check the infeasibility of the problem. For example, this strategy can be used for symmetry constraints over integer variables. The symmetry constraints are very helpful to prune the search tree when using the FD propagation, but they are not necessary for simplex to recognise an infeasibility.

3.3 Constraints for the CLP and/or MIP solver

We modify the proposed translator to specify which constraints should be considered by the MIP solver, the CLP solver or by both solvers. Such a translation can be useful for solving the problems which represent a combination of different types of problems. An example is the Video Broadcast Service (VBS) problem [25] which is a combination of scheduling and broadcast network routing. The empirical results show that the problem is tackled better if the MIP solver is only used on the routing and CLP on the scheduling part of the problem [24].

We define two stores of constraints. The *CLP-store* represents the CLP part of the model and the *MIP-store* represents the MIP part. The whole model of the problem is then represented by both stores:

$$Model \;=\; (\text{CLP-store, MIP-store}).$$

To denote which constraints should be handled by a particular solver, new types of the predicate symbols are introduced to the language. Every linear constraint with predicate symbol $\# \leq$ is put into the CLP store and every linear constraint with the predicate symbol $\$\leq$ is put into the MIP store.

Replacement 4: *A constraint $X\#\$\leq Y$ is replaced by*

$$\texttt{X\#} \leq \texttt{Y},\ \texttt{X\$} \leq \texttt{Y}.$$

Replacement 4 allows the user to send a constraint to the both solvers. The proposed extensions of the CLP language in this section leads to a new definition of a program. A *program P* is redefined by a set of clauses and rules such that the heads of clauses do not have common predicate symbols with the heads of rules. A *generic program gen*(P) is a program with the rules and translated clauses of P using the defined directives and replacements. We show that the translation of such redefined programs preserves the set of all solutions.

Theorem 3.1
If $p(Args)$ is a goal in the language of a program P with rules and clauses, then it has the same set of solutions when evaluated against P and against $gen(P)$.

Proof
Every predicate is defined by clauses or by rules or it is already a constraint. If it is defined by clauses, then the result follows by Lemma 2.2. Otherwise, using the operational semantics of the commit operator, the constraint commits exactly one rule and it does not change the set of solutions of the translated program. □

Let us demonstrate the benefit of program rules by an example:

Example 3.2
A program to check if the difference between ANY two numbers of a given set is greater than or equal to two can be defined as follows:

```
test_all(V)  ⇔  V = [] | true.
test_all(V)  ⇔  V = [X|R] | binary(X), 1 ≤ X, X ≤ 10, test(X,R), test_all(R).
  test(X,L)  ⇔  L = [] | true.
  test(X,L)  ⇔  L = [Y|R] | diff(X,Y), test(X,R).
  diff(X,Y)  :-  Y + 2 ≤ X.
  diff(X,Y)  :-  X + 2 ≤ Y.
```

The program is translated to the generic program:

```
test_all(V)  ⇔  V = [] | true.
test_all(V)  ⇔  V = [X|R] | binary(X), 1 ≤ X, X ≤ 10, test(X,R), test_all(R).
  test(X,L)  ⇔  L = [] | true.
  test(X,L)  ⇔  L = [Y|R] | diff(X,Y), test(X,R).
```

$$\begin{aligned}
\mathtt{diff(X,Y)} &:- \quad \mathtt{diff(X,Y,1)}.\\
\mathtt{diff(X,Y,B)} &:- \quad \mathtt{diff_1(X,Y,B_1),\ diff_2(X,Y,B_2),\ B_1 + B_2 = B}.\\
\mathtt{diff_1(X,Y,B_1)} &:- \quad \mathtt{Y + 2 + 11 * B_1 \ \le\ X + 11}.\\
\mathtt{diff_2(X,Y,B_2)} &:- \quad \mathtt{X + 2 + 11 * B_2 \ \le\ Y + 11}.
\end{aligned}$$

For a set V with three variables X, Y and Z, the generic model is unfolded into type and linear constraints:

$$\begin{array}{l}
\mathtt{integer(X),\ 1 \le X,\ X \le 10,}\\
\mathtt{integer(Y),\ 1 \le Y,\ Y \le 10,}\\
\mathtt{integer(Z),\ 1 \le Z,\ Z \le 10,}\\
\mathtt{Y + 11 * B_{11} \ \le\ X + 9,\ \ X + 11 * B_{12} \ \le\ Y + 9,\ \ B_{11} + B_{12} = 1,}\\
\mathtt{Z + 11 * B_{21} \ \le\ X + 9,\ \ X + 11 * B_{22} \ \le\ Z + 9,\ \ B_{21} + B_{22} = 1,}\\
\mathtt{Z + 11 * B_{31} \ \le\ Y + 9,\ \ Y + 11 * B_{32} \ \le\ Z + 9,\ \ B_{31} + B_{32} = 1.}
\end{array}$$

□

4 Combining MIP and CLP Solvers

The second part of our paper represents an integration of the CLP solver and the MIP solver by using the generated model as the result of the proposed translation in the previous sections. The integration is based on combining efficient components of both solvers: the finite domain handling as a local constraint propagation by CLP, and simplex as a global constraint propagation by MIP. Using ECLiPSe to control the search and applying both constraint propagations on decisions during the search, our approach is based on the following *constructive search strategies*:

CONSTRUCTIVE SEARCH STRATEGIES

Step 0 Unfold the set of constraints from the generic model of a given problem.

Step 1 Control the search by ECLiPSe using the local constraint propagation on finite domains and the global constraint propagation by simplex on the continuous relaxed problem.

In this section we show that our approach is different to other solution techniques and that the CLP solver and the MIP solver are only instances of the constructive search strategies. Furthermore, we present a constructive search strategy combining both solvers and show the solution results on an example.

4.1 Comparison to other solvers

The techniques on combining CLP and MIP solvers can be split into three groups.

The first group represents techniques which are based on the CLP solver strategy. Backer and Beringer [1] have shown how to use a linear relaxation of disjunctive constraints to reduce the solution space of problems having a natural "geometric" formulation. There are also some techniques on how to use linear algebraic forms on some classes of constraints. For example, Barth and Bockmayer [2] investigated the class of pseudo-Boolean constraints while Hooker [14] generalised resolution for linear inequalities involving only 0-1 variables.

The second group consists of techniques which are based on the MIP solver strategy. Little and Darby-Dowman [20] discussed the ability to improve the MIP solver by adding local cuts as a form of intra-processing (not pre or post but carried out during the MIP search). Hoffman and Padberg [15] demonstrated the Branch and Cut technique on large airline crew-scheduling problems.

The third group represents techniques which use the CLP and MIP solver to solve the problem. Hajian et. al. [11] demonstrated how to use a feasible solution from the CLP solver to "warm-start" the MIP solver on the Fleet Assignment Problem.

We combine the CLP and MIP solvers using the local constraint propagation on finite domains and the global constraint propagation using simplex on the continuous relaxed problem. The basis of the CLP and MIP solvers is in fact quite close; choice of variable and choice of value, but in CLP, the choice of value comes from an enumeration process while MIP uses the value as a part of an imposed constraint. In particularly, when dealing with binary variables in an MIP model, the branching in the search tree corresponds precisely to the assignment of 0 or 1 to a chosen variable.

Furthermore, CLP can control the search not only by assigning values to variables and adding new bounds, but also by identifying specific heuristics used for this problem. It is also possible to use the experts experience in how they would go about solving the problem or where to look for an optimal solution. Notice that the tree search in MIP is based on the fixed strategies provided by the MIP solver.

Consider the set of auxiliary variables which are generated by the translation: the same fine control over the execution of the original program can be produced as it is given by the implicit enumeration of the default evaluation algorithm. The search tree of the original program can be reproduced by labelling on the auxiliary variables of the translated program. Since the translated linear constraints represent the whole problem the problem can be solved by only the MIP solver, or by only the CLP solver, or by hybrid algorithms which combine both solvers.

In the following we represent three instances of the constructive search strategies: the CLP solver, the MIP solver and an hybrid algorithm. We show that the hybrid algorithm has the good characteristics of both solvers.

4.2 Constructive search using local constraint propagation: the CLP solver

If we use only local constraint propagation (e.g., the propagation on finite domains), then the constructive search strategy characterises the *CLP solver*: a solution is found through enumeration, where values from the domains are assigned to the variables, while ensuring that the constraints are continually satisfied, until all the variables have been given a value [9]. Enumeration generates a search tree. Constraints can force a variable to have an empty domain, indicating that no value is possible, given the current assignments, satisfying all the constraints. Therefore, enumeration in such a branch of the tree is stopped. Then backtracking takes place and during this process, variable bindings and domain reductions will be undone automatically as they do not lead to a feasible solution.

Within the same solution framework, it is possible to go beyond a first feasible solution to finding an optimal one. The process of finding an optimal solution fits into the CLP solver method by adding a new cost constraint each time a better feasible solution is found. Better solutions, if they exist, are repeatedly found and appropriate tighter constraints added until the whole solution space has been effectively covered. This is a CLP implementation of the Branch and Bound technique, applied to reduce the amount of searching required by identifying as early as possible parts of the search tree which will not lead to a better cost.

The algorithm below is an algorithm of the CLP solver which will be used on our test problems in the following section.

CONSTRUCTIVE SEARCH USING ONLY LOCAL CONSTRAINT PROPAGATION [10]
(the CLP solver)

Step 0 Unfold the set of constraints from the generic model of a given problem.

Step 1 Choose an unlabelled most-constrained variable v in the underlying structure of the problem.

Step 2 Choose a new value to v.

Step 3 Update the cost and the domains of other variables using the local constraint propagation on finite domains. If an infeasibility is recognised, then backtrack to Step 2. If all variables are instantiated and the solution has the minimal cost, then the algorithm terminates, else go to Step 1.

The key to the performance of the CLP solver is determined by three factors. The first is the number and the size of the finite domain variables introduced in the model. The solution space is usually kept as small as possible. The second factor in determining performance is the choice and number of constraints. Generally, the more constraints, the greater the search reduction which can take place. The third factor is in the search strategy. The routing of the search corresponds precisely to the choice of next variable and the choice of value to assign. The strategies are often based on mathematical characteristics of the domains at that time, e.g., to select the domain variable with the smallest domain or the variable which has most constraints attached to it.

4.3 Constructive search using global constraint propagation: the MIP solver

Global constraint propagation is represented by solving the continuous relaxed problem using, e.g., the simplex algorithm. The relaxed problem is simply defined by replacing *integer* and *binary* type constraints in the MIP store by *real* type constraints. Instead of the implementation of simplex in ECLiPSe, the linear constraints of the MIP store are put to CPLEX. When the global propagation is called, ECLiPSe sends bounds of variables to CPLEX and simplex returns an optimal solution of the relaxed problem. The solution of the relaxed problem helps to solve the whole problem using, e.g., the Branch and Bound technique for MIP. This technique was established as the dominant solution technique for solving discrete optimisation problems [18].

The solution method for Branch and Bound involves checking whether the solution becomes integer given successive splitting of the solution space. This segmentation of the solution space is achieved by selecting one of the violating variables v from the relaxed problem, e.g., a variable which has a fractional value but is required to be integer. Two new subproblems are generated by adding inequalities $v \geq \lceil s(v) \rceil$ and $v \leq \lfloor s(v) \rfloor$ to the original problem, respectively. In case that the chosen variable is a binary variable, then the two new subproblems are generated by fixing the chosen variable to 1 in one and to 0 in the other subproblem. A search tree is built by selecting one of the violated variables to branch on and chose a subproblem to relax and solve. The tree search continues to seek optimality, but any node in the tree for which the relaxed problem gives a solution worse than the current best integer solution, is discarded along with that branch.

The algorithm below is an algorithm of the MIP solver which will be used on our test problems in the following section.

CONSTRUCTIVE SEARCH USING GLOBAL CONSTRAINT PROPAGATION ONLY [8]
(the MIP solver)

Step 0 Unfold the set of constraints from the generic model of a given problem. If there is no unique type constraint for each variable, then the algorithm reports an error. Let s be the solution of the continuous relaxed problem using simplex. If s satisfies all the integer restrictions, then it is the optimal solution of the problem, and the algorithm terminates. If an infeasibility is recognised, then the problem does not have a solution and the algorithm terminates in failure.

Step 1 Choose a variable v with a non-integer solution value $s(v)$.

Step 2 Split the problem into two subproblems by adding the linear constraints $v \geq \lceil s(v) \rceil$ and $v \leq \lfloor s(v) \rfloor$, respectively. Choose one of these subproblems.

Step 3 Compute the solution s of the continuous relaxed subproblem by using simplex. If an infeasibility is recognised, then backtrack to Step 2 and choose the other subproblem. If every variable v has an integer solution value $s(v)$ and the solution has a minimal cost, then the algorithm terminates. If there is a variable with a non-integer value $s(v)$, then go to Step 1 else go to Step 2 and choose the other subproblem.

The search strategy in terms of variable choice and branch choice is important in determining how the tree develops and consequently how quickly the first feasible solution is found, how good it is and what is the time taken to find the optimal solution. It is often the case that different search strategies are built into the MIP software. These are based on the choice of variable and branch according to mathematical characteristics of the solution tree at that point. The strategies can be, for example, to select the variable with a value of the relaxed problem closest to an integer number and the highest cost value. There can be a wide variation in performance of a strategy from one problem to another [8]. Therfore, the most suitable strategy can be derived by trying all possible strategies. However, [8] categorises the problem such that a suitable strategy can be chosen with regard to a specific problem structure in advance.

4.4 Constructive search using local and global constraint propagation: a hybrid CLP & MIP solver

The main idea of our integration of MIP with CLP is to make decisions as late as possible. Using local and global constraint propagation the solution space of a problem can be reduced more than by using one type of propagation only. In the hybrid algorithm, the search is controlled by allowing a dynamic choice of variable and value ordering, and applying simplex as well as the finite domain propagation at each node in the search tree. If simplex recognises an infeasibility, then the CLP solver does not need to search for a feasible solution, which is done by performing only the CLP solver.

CONSTRUCTIVE SEARCH USING LOCAL AND GLOBAL CONSTRAINT PROPAGATION

Step 0 Unfold the set of constraints from the generic model of a given problem. If there is no unique type constraint for each variable, then the algorithm reports an error. Let s be the solution of the continuous relaxed problem using simplex. If s satisfies all the integer restrictions, then it is the optimal solution of the problem, and the algorithm terminates. If an infeasibility is recognised, then the problem does not have a solution and the algorithm terminates in failure.

Step 1 Choose an unlabelled most-constrained variable v in the underlying structure of the problem.

Step 2 Choose the nearest integer value to $s(v)$ for variable v.

Step 3 Update the cost and the domains of other variables using local constraint propagation on finite domains. If an infeasibility is recognised, then go to Step 4. If all variables are instantiated, then go to Step 3a, else go to Step 1.

Step 3a If the solution has the minimal cost, then the algorithm terminates, else go to Step 4.

Step 4 Choose a new value for v, otherwise backtrack to Step 1.

Step 4a Compute the solution s of the continuous relaxed subproblem using simplex. If an infeasibility is recognised, then go to Step 4, else go to Step 2.

In Step 2, the algorithm instantiates the most-constrained [9] variable v by using the nearest integer number to the real value $s(v)$, where s is the optimal solution of the relaxed problem. If the instantiation causes an infeasibility by applying the local constraint propagation, then variable v is instantiated in Step 4 by another value. In this situation simplex is performed. In the worst case, the algorithm can require more computation time than required by the MIP or CLP solver alone. A comparison between the MIP, CLP and CLP&MIP solvers is given on four problems in the following section.

5 Computational results on different problems

In this section we present the empirical results of the proposed integration MIP with CLP on different problems. For each problem we present the time to derive the first feasible solution, the time to derive an optimal solution and the time to prove its optimality. We show that the combination of both solvers derives the optimal solution and proves optimality to each problem in reasonable time, while the CLP solver and the MIP solver are not able to achieve the same performance. The problems are defined in the next subsection and the result on modelling and solving them are represented in the following two subsections. Finally, we compare the results of different solvers on different models.

5.1 Problems

We consider four problems which are difficult for the CLP or MIP solver.

1. *k-Hoist Scheduling Problem [23]:* The problem, denoted by k-HSP, occurs when a component must be allotted a sequence of tanks during its manufacture. The component must remain in these tanks for periods of time lying within specific bounds. k hoists are to be programmed to place components into tanks, remove components from tanks, and transport components between tanks, so as to maximise the throughput of the production line.

2. *Progressive Party Problem [27]:* The problem, denoted by PPP, arises in the context of organising a "progressive party" at a yachting rally with 42 yachts. Some yachts are to be designated hosts; the crews of the remaining yachts visit the hosts for six successive half-hour periods. A guest crew cannot revisit the same host, and two guest crews cannot meet more than once. Additional constraints are imposed by the capacities of the host yachts and the crew sizes of the guests. The problem is to minimise the number of host boats.

3. *Cabinet Assignment Problem [20]:* The problem, denoted by CAP, arises in the context of producing specified numbers of different types of telecommunication cabinets over a fixed time period. Each cabinet type requires a different set of elementary sequential operations, each taking the same time to be carried out. The manufacturing process involves a number of identical unit cells linked together to form a pipeline machine capable of carrying out the tasks. Each unit cell is capable of executing all operations to build any particular type of cabinet. A machine with a particular number of cells is allowed only for manufacturing cabinets when the number of operations is a multiple of the number of cells. For example, a two-cell machine is only suitable for building cabinet types with 2, 4, 6, ... separate operations. The problem is to allocate each task to a machine such that the total production is completed as early as possible.

4. *Set Partitioning Problem [15]:* The problem, denoted by SPP, is to collect a set of M subsets of N, $M \subseteq 2^N$, such that they are pairwise disjoint, their union is N, and the sum of the weights of the subsets is minimal. (We consider the example with 197 subsets over 17 elements [15].)

These benchmarks are chosen for a number of reasons:

Phillips and Unger [23] have demonstrated that the MIP solver derives the optimal solution to the 1-Hoist Scheduling Problem very quickly. Unfortunately, the efficiency quickly diminishes as the number of tanks and hoists in a given system increases. Scheduling two or more hoists expands the solution space, and makes the task of searching for the global optimal solution extremely difficult. For instance, Lei and Wang [19] have proposed a heuristic algorithm for the 2-Hoist Scheduling Problem to derive feasible solutions without proof of optimality.

Smith et al. [27] have shown that the difference in the performance of the CLP solver versus the MIP solver on the Progressive Party Problem is particularly marked. The MIP model is too large to be solved using linear programming techniques, whereas constraint programming can succeed through careful choice of heuristics to direct its search. Although a solution with 13 hosts is derived quickly, constraint programming has difficulties in recognising the optimal solution. On the other hand, simplex easily recognises the infeasibility for 12 hosts.

Little and Darby-Dowman [20] have shown that the CLP solver solves the Cabinet Assignment Problem quicker than the MIP solver. The MIP solver reaches an integer solution equal to the optimum, but is not able to prove its optimality.

Hoffman and Padberg [15] have demonstrated that the MIP solver solves the Set Partitioning Problem very efficiently. The difference in the performance of the CLP solver versus the MIP solver on this problem is particularly marked.

5.2 Modelling

Problems in CLP are modelled in terms of finite domain variables and the constraints over them. Almost all of the mentioned problems are represented in a very natural way using non-binary variables. By performing the proposed translation of CLP programs, two sets of constraints are derived: a set of constraints representing the CLP store, and a set of linear constraints representing the MIP store. We show three different models for each problem:

1. the CLP store of constraints
2. the MIP store of translated constraints
3. the CLP store and the MIP store.

To perform a hybrid algorithm, both CLP and MIP stores are generated. Each of the stores does not need to represent the whole problem. For instance, the CLP store can define the whole problem while the MIP store can contain only some translated linear constraints.

The number of constraints and variables of different models for the problems are given in Tables 1-3. Table 1 gives the number of all variables, all constraints, only disjunctive constraints and only binary variables of the CLP store for the CLP solver. Tables 2 and 3 represent the number of rows, columns, non-zeros and binary variables of the mathematical model in CPLEX after crossing the linear constraints from the MIP store. In Table 2, the MIP store is the result of the automatic translation using auxiliary variables.

Modelling of the k-Hoist Scheduling Problem: The problem can be represented by disjunctive constraints over non-binary variables. For every two tanks i and j, the following disjunctive constraints are defined:

$$\begin{aligned}
\mathtt{disj(H_i,H_j,T_i,T_j,T)} &\ :- \quad \mathtt{T_i + f(i) + e(i+1,j) \leq T_j + k * T.}\\
\mathtt{disj(H_i,H_j,T_i,T_j,T)} &\ :- \quad \mathtt{T_j + k * T + f(j) + e(j+1,i) \leq T_i.}\\
\mathtt{disj(H_i,H_j,T_i,T_j,T)} &\ :- \quad \mathtt{H_i + 1 \leq H_j.}\\
\mathtt{disj(H_i,H_j,T_i,T_j,T)} &\ :- \quad \mathtt{H_j + 1 \leq H_i.}
\end{aligned}$$

Variables H_i and H_j represent the hoists which transport components from tank i and tank j, respectively. Variables T_i and T_j represent the removal time of the components from tank i and tank j, respectively. Variable T represents the period of the the production line. The goal of the problem is to minimise this variable. Constant $f(i)$ denotes the transport time of the components from tank i to the next tank $i+1$. Constant $e(i,j)$ denotes the time of hoists (when empty) to move from tank i to tank j. By unfolding the CLP program, the constraints of the problem are stored into the the CLP store and the MIP store (see Tables 1-3 for the 2-Hoist Scheduling Problem).

Modelling of the Progressive Party Problem: The problem can be represented by disjunctive constraints over non-binary variables (using `alldistinct` constraint), capacity constraints over binary variables (i.e., linear equalities and inequalities) and link constraints to link binary variables and non-binary variables (using `link` constraint). Since the problem has many equivalent solutions such symmetries in the problem can vastly increase the size of the search space. Symmetry is avoided, or at least reduced, by adding symmetry constraints to eliminate equivalent solutions.

By unfolding the CLP program for this problem, all mentioned constraints are derived and stored into the CLP store (see Table 1). This model is appropriate for the CLP solver. A model for the MIP solver is derived by using our translator (see Table 2). The main reason for a large number of translated constraints and variables for the Progressive Party Problem is the translation of link constraints between binary and non-binary variables.

The hybrid CLP&MIP solver is performed on a model which is derived by using the replaced and translation directives on the disjunction, capacity and link constraints, which are sufficient to define the problem. The symmetry constraints are helpful for the CLP solver but they are usually not needed for the MIP solver. We do not translate them. The disjunctive constraints are replaced by linear constraints on binary variables. Since there are no other constraints over non-binary variables we did not need to translate the constraints linking binary variables to non-binary variables. The set of translated constraints becomes much smaller than the translation of the whole program (see Table 3).

Modelling of the Cabinet Assignment Problem: All three models are generated on the same way as the models of the Progressive Party Problem (see Tables 1-3).

Modelling of the Set Partitioning Problem: The whole problem is represented by constraints over binary variables. All three models are equivalent since the constraints are linear equalities over binary variables.

Table 1: The CLP store of the CLP models.

CLP store	2-HSP	PPP	CAP	SPP
Vars.	507	4806	98	198
Constr.	974	5861	118	18
Disj. constr.	641	5281	97	184
Bin. vars.	493	4632	84	198

Table 2: The MIP store of the MIP models.

MIP store	2-HSP	PPP	CAP	SPP
Rows	1328	14592	294	198
Columns	2146	24593	511	18
Non-zeros	7861	108134	1158	184
Bin. vars.	1314	14418	280	198

Table 3: The MIP store of the CLP&MIP models.

MIP store	2-HSP	PPP	CAP	SPP
Rows	1328	4632	84	198
Columns	2146	1035	109	18
Non-zeros	7861	7980	530	184
Bin. vars.	1314	4632	84	198

5.3 Solving

We have implemented the integration of CLP with MIP by using the ECLiPSe constraint logic programming platform and the CPLEX mathematical programming package. This allows CPLEX to be used to solve problems modelled in ECLiPSe. The control of the search process and the local constraint propagation is handled by CLP, while the global constraint propagation is handled by MIP. The local propagation is performed by a consistency algorithm on finite domains and it represents a component of the ECLiPSe package. On the other hand, the global propagation is performed by the simplex algorithm which is a component of the CPLEX package. We apply the following tasks:

- the CLP solver on the original CLP constraints (see Table 1)
- the MIP solver on the translated CLP constraints (see Table 2)
- the hybrid CLP&MIP solver on the original CLP constraints (see Table 1) and on the partially translated CLP constraints (see Table 3).

Let us discuss the empirical results of the hybrid solver relative to the results of the CLP and MIP solvers on each problem. We say that a problem is solved by a solver if this solver derives an optimal solution to the problem and proves its optimality. The results show that the Cabinet Assignment Problem is easy for the CLP solver and hard for the MIP solver. The Set Partitioning Problem is hard for the CLP solver and easy for the MIP solver. The 2-Hoist Scheduling Problem and the Progressive Party Problem are hard for the CLP solver and for the MIP solver. However, all test problems are easy for the hybrid solver. The empirical results in Table 4 demonstrate that the hybrid algorithm derives an optimal solution and proves its optimality to each problem, which cannot be achieved by the CLP solver or the MIP solver. All times are in CPU seconds running on a SUN-SPARC/20.

Solving the 2-Hoist Scheduling Problem: The CLP model contains 974 constraints over 507 variables and the CLP solver does not derive a solution within 5 minutes. The MIP solver on the translated CLP model is also an inefficient approach to solve this problem. By applying the CLP&MIP solver on the CLP store (see Table 1) and on the MIP store (see Table 3), simplex and the constraint propagation on finite domains helped to derive an optimal solution and to prove its optimality in 102.65 sec. It follows that the local and global constraint propagation are very useful procedures, cutting the solution space and deriving an optimal solution to the hoist problem in reasonable time.

Solving the Progressive Party Problem: The CLP model contains 5861 constraints over 4806 variables and the CLP solver derived a solution with 13 hosts in 171.03 sec. This was an optimal solution of the problem, but the CLP solver did not recognise it within two hours. By translating the program, the MIP store contains 24593 constraints over 14592 variables. The MIP solver did not succeed in deriving a solution with 13 hosts within two hours. It needed 509 sec. to even recognise an infeasibility for 12 hosts.

By applying the CLP&MIP solver on the CLP store (see Table 1) and the MIP store (see Table 3), the local propagation on finite domains helped to derive a solution with 13 hosts, while simplex helped to recognise infeasibility for 12 hosts. An optimal solution of the problem was derived in 211.69 sec.

It follows that the CLP solver is good at deriving a solution with 13 hosts while the MIP solver is good at recognising infeasibility for 12 hosts and, hence, at recognising that the solution of the CLP solver is an optimal solution. Furthermore, the proposed translation of the whole problem can lead to a very inefficient simplex algorithm. It confirms the claim that putting effort on the modelling part is indeed a step towards an efficient running program.

Solving the Cabinet Assignment Problem: The CLP solver derives an optimal solution while the MIP solver finds a solution with the optimal cost but it has difficulties in proving optimality. By applying the CLP&MIP solver, the number of backtrack steps was reduced from 287 to 122 w.r.t. the CLP solver. This shows that simplex helps a lot to reduce the number of decisions and, hence, the size of the search tree.

Table 4: Characteristics of the solvers on different models.

The CLP Solver on the CLP store:

	2-HSP	PPP	CAP	SPP
Time (1st solution)	$>$ 5 min	24.51	2.15	42.38
Time (optimal solution)	$>$ 5 min	171.03	8.24	59.93
Time (proof of optimality)	$>$ 5 min	$>$ 5 min	9.27	64.68
Best Solution	-	13	55.8	11307
FD fails	$>$ 15000	$>$ 15000	287	1611

The MIP Solver on the MIP store:

	2-HSP	PPP	CAP	SPP
Time (1st solution)	$>$ 5 min	$>$ 5 min	0.34	0.15
Time (optimal solution)	$>$ 5 min	$>$ 5 min	5.12	0.18
Time (proof of optimality)	$>$ 5 min	$>$ 5 min	$>$ 5 min	0.18
Best Solution	-	-	55.8	11307
Nodes processed	$>$ 15000	$>$ 15000	400	4

The CLP&MIP Solver:

	2-HSP	PPP	CAP	SPP
Time (1st solution)	20.40	38.51	7.31	0.71
Time (optimal solution)	41.23	211.69	10.52	0.71
Time (proof of optimality)	102.65	303.23	13.21	0.71
Best Solution	251	13	55.8	11307
FD fails	2377	3283	122	0
LP fails	1598	5327	63	0

Solving the Set Partitioning Problem: This problem is solved faster using the MIP solver which generated only 4 nodes to derive an optimal solution in 0.18 sec. For instance, simplex derived a relaxed optimal solution with 172 integer values and only 4 non-integer values. On the other hand, the CLP solver needed 64.68 sec to solve the problem. By applying the CLP&MIP solver, an optimal solution was derived without backtracking. Simplex was called only once, and by using the suggested integer values, the local propagation on finite domains causes an instantiation of integer values for all other variables.

5.4 A comparison: the CLP solver on the translated CLP model versus the MIP solver on the translated CLP model

The proposed automatic translation does not guarantee the "best" model either for the MIP solver or for the CLP solver. An example of such a model for the MIP solver is the MIP store of the translated CLP constraints for the Progressive Party Problem (see Table 2). The main reason for such a large number of constraints and variables was the automatic translation of the link constraints between binary and non-binary variables.

By applying the CLP solver on the MIP store of the translated CLP constraints, the local constraint propagation is significantly less efficient when reducing the domains of variables. Let us demonstrate the inefficiency on disjunctive constraint $(X + 2 \leq Y \textit{ or } Y + 2 \leq X)$, where X and Y are integer variables with the domain 0..10. If the CLP solver chooses a value for variable X, e.g., $X = 5$, then the finite domain propagation on the disjunctive constraint reduces the domain of Y to $\{0, ..., 3, 7, ..., 10\}$, while it does not modify the domain of Y by considering the translated constraints.

5.5 A comparison: the CLP solver on the pure CLP model versus the hybrid CLP&MIP solver on the partially translated CLP model

The empirical results have shown that the hybrid CLP&MIP solver derives optimal solutions faster if the CLP constraints are replaced and translated to linear constraints over binary variables. The link constraints and symmetry constraints are not translated. This generally improves performance, most significantly finding optimality within the limited number of decisions. However, the optimal solution is not necessarily derived faster than when using the CLP solver. For example, the results on the Progressive Party Problem and the Cabinet Assignment Problem have shown that the times were still inferior to the total CLP approach. On the other problems, the hybrid algorithm was faster than the CLP solver since simplex effectively reduced the solution space by solving the relaxed problem. It seems that if a problem has not been identified as being suitable for an CLP solution, the decision of which hybrid algorithms is most adequate is still unclear.

5.6 The best and the worst performance in relation to the problem characteristics

We have presented problems which are relatively easy for the CLP solver and hard for the MIP solver (e.g., the Cabinet Assignment Problem), as well as problems which are hard for the CLP solver and easy for the MIP solver (e.g., the Set Partitioning Problem). The empirical results show that the hybrid CLP&MIP solver successfully derives an optimal solution and proves its optimality in all problems.

The structure of a given problem can help to determine when CLP should be chosen in preference to MIP. All problems in this section, except the k-Hoist Scheduling Problem, have the common characteristic that they contain `alldistinct` constraints and *capacity* linear constraints (i.e., $\sum_i c_i v_i \leq C$). The empirical results demonstrate that the CLP solver is faster than the MIP solver if values c_i, $1 \leq i \leq n$, are different (e.g., the Progressive Party Problem and the Cabinet Assignment Problem).

6 Conclusions

The CLP modelling ensures that the encoding of a correct model of the problem can indeed be a step towards an efficient running program. CLP allows the definition of constraints in a more natural and compact way. If CLP is also used to control the search by using the local and global constraint propagation, the consequences can be revolutionary - with programmers actually taking modelling seriously.

We have presented an efficient translation to derive a generic model and further specific MIP models for different input data. The translation is performed only on the part of the program not satisfying data/control-flow properties. The translated program enables us not only to use different solution techniques (the CLP solver or the MIP solver), but also to combine them into more powerful mechanisms (hybrid CLP&MIP solvers). The results of such an integration of two programming paradigms are represented by the Progressive Party Problem and the 2-Hoist Scheduling Problem, which can be solved efficiently, while the CLP solver or the MIP solver are not able to solve them in reasonable time. The integration allows comparisons between the two approaches to give a clearer idea of when CLP should be chosen in preference to MIP, and when an integrated solver is faster than the CLP solver or the MIP solver.

References

[1] B. deBacker and H. Beringer, A CLP language handling disjunctions of linear constraints, in: *Proceedings of the ICLP*, Budapest, 1993, pp. 550-563.

[2] P. Barth and A. Bockmayer, Modelling mixed-integer optimisation problems in constraint logic programming, Technical Report, MPI-I-95-2-011, Max-Planck-Institut für Informatik, Saarbrücken, 1995.

[3] N. L. Biggs, *Discrete Mathematics*, Oxford Science Publications, 1994.

[4] I. Bratko, *Prolog Programming for Artificial Intelligence*, Addison-Wesley Publishing, 1990.

[5] CPLEX, Using the CPLEX callable library, Ver3, CPLEX Optimisation Inc., Suite 279, Tahoe Blvd, Bldg. 802, Incline Village, NV 89451-9436, 1995.

[6] J. David and C. Tat-Leong, Constraint-based applications in production planning: examples from the automative industry, in: *Proceedings of the Practical Application of Constraint Technology*, Paris, 1995, pp. 37-51.

[7] ECL^iPS^e 3.5 user manual, ECRC, Munich, 1995.

[8] M. T. Hajian, Computational methods for discrete programming problems, PhD Thesis, Department of Mathematics and Statistics, Brunel University, Uxbridge, UK, 1993.

[9] P. V. Hentenryck, *Constraint Satisfaction in Logic Programming*, Logic Programming Series, MIT Press, Cambridge, 1989.

[10] P. V. Hentenryck, Constraint logic programming, The Knowledge Engineering Review 6(3) (1991) 151-194.

[11] M. T. Hajian, H. El-Sakkout, M. G. Wallace, J. M. Lever and E. B. Richards, Towards a closer integration of finite domain propagation and simplex-based algorithm, Technical Report ICPARC-95/09-01, IC-Parc, Imperial College, London, 1995.

[12] E. Hadjiconstantinou, C. Lucas, G. Mitra and S. Moody, Tools for reformulating logical forms into zero-one mixed integer programs, European Journal of Operational Research 72(2) (1994) 262-276.

[13] C. Holzbaur, A specialized, incremental solved form algorithm for systems of linear inequalities, Technical Report, TR-94-07, Austrian Research Institute for Artificial Intelligence, Vienna, 1994.

[14] J. N. Hooker, Generalized resolution for 0-1 linear inequalities, Annals of Mathematics and Artificial Intelligence 6 (1992) 271-286.

[15] K. L. Hoffman and M. Padberg, Solving airline crew-scheduling problems by branch-and-cut, Technical Report, George Mason University and New York University, USA, 1992.

[16] K. I. M. McKinnon and H. P. Williams, Constructing integer programming model by the predicate calculus, Annals of Operations Research 21 (1989) 227-246.

[17] A. D. Kelly, A. Macdonald, K. Mariott, H. Sondergaard, P. J. Stuckey and R. H. C. Yap, An optimizing compiler for CLP(R), in: *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, Cassis, 1995, pp. 222-239.

[18] A. Land and A. Doig, An automatic method for solving discrete programming problems, Econometrica 28(3) (1960) 497-520.

[19] L. Lei and T. J. Wang, The minimum common cycle algorithm for cycle scheduling of two material handling hoists with time window constraint, Management Science 37(12) (1991) 1629-1639.

[20] J. Little and K. Darby-Dowman, The significance of constraint logic programming to operational research, Technical Report, Brunel University, Department of Mathematics and Statistics, Brunel University, Uxbridge, UK, 1995.

[21] M. J. Maher, Logic semantics for a class of committed-choice programs, in: *Proceedings of the ICLP*, Melbourne, 1987, pp. 858-876.

[22] C. L. Pape, Implementation of resource constraints in ILOG scheduling: a library of the development of constraint-based scheduling systems, Intelligent Systems Engineering 3(2) (1994) 55-66.

[23] L. W. Phillips and P. S. Unger, Mathematical programming solution of a hoist scheduling program, AIIE Transactions 8(2) (1976) 219-225.

[24] D. Pothos, Broadcast network routing using constraint logic programming, Technical Report, IC-Parc, Imperial College, London, 1995.

[25] B. Purohit, T. Clark and T. Richards, Techniques for routing and scheduling services on a transmission network, BT Technology Journal 13(1) (1995) 64-72.

[26] J. Puget, A comparison between constraint programming and integer programming, in: *Proceedings of the Applied Mathematical Programming and Modelling Conference*, Brunel University, Uxbridge, UK, 1995.

[27] B. M. Smith, S. C. Brailsford, P. M. Hubbard and H. P. Williams, The progressive party problem: integer linear programming and constraint programming compared, in: *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, Cassis, 1995.

[28] M. Wallace, Applying constraints for scheduling, *Constraint Programming, NATO ASI Series*, eds. B. Mayoh, E. Tyugu and J. Penjam, Springer-Verlag, 1994, pp. 153-172.

[29] H. P. Williams, Logic problems and integer programming, Bulletin of the Institute of Mathematics and its Applications 13 (1977) 18-20.

[30] H. P. Williams, Linear and integer programming applied to the propositional calculus, International Journal of Systems Research and Information Science 2 (1987) 81-100.

[31] H. P. Williams, *Model Building in Mathematical Programming*, John Wiley and Sons, 1990.

[32] H. P. Williams, *Model Solving in Mathematical Programming*, John Wiley and Sons, 1993.