Eplex: Harnessing Mathematical Programming Solvers for Constraint Logic Programming

Kish Shen and Joachim Schimpf

IC-Parc, Imperial College London, London SW7 2AZ, United Kingdom
{k.shen,j.schimpf}@icparc.ic.ac.uk

Abstract. The *eplex* library of the ECL^{i}PS^e Constraint Logic Programming platform allows the integration of Mathematical Programming techniques with its native Constraint Logic Programming techniques within the same unified framework. It provides an interface to state-of-the-art Mathematical Programming solvers, and a set of programming primitives that allow 'hybrid' techniques to be easily expressed. This paper presents these facilities, and discusses some associated implementation issues.

1 Introduction

Constraint Programming (CP) and Mathematical Programming (MP) are two approaches that have been used to tackle large scale Combinatorial Optimisation problems. In recent years, there has been significant research effort [19] to combine the two, exploiting their complementary strengths, to develop 'hybrid' algorithms that can tackle problems that are difficult for either approach alone.

Much of the recent algorithmic research and development work at IC-Parc has been focused on this hybrid approach. The ECL^{i}PS^e Constraint Logic Programming (CLP) platform is the programming environment used for this development work. As developers of ECL^{i}PS^e, we aim to provide a unified high-level platform for programmers to explore different approaches to solving their problems. To allow the exploration of hybrid and MP techniques from a CLP perspective, we developed the *eplex* library for ECL^{i}PS^e, whose first version was released in 1997 and which has been under continuous development since, mainly driven by application requirements.

In this paper, we present the *eplex* library in its current form. Our aim is to highlight the facilities provided which enable the MP/hybrid algorithms to be developed in ECL^{i}PS^e, rather than describing the algorithms themselves. We also assume that the reader has some familiarity with CLP languages and concepts.

2 Motivation and Objectives

With this work, we pursued the following objectives

- make the convenience of CLP available for modelling MP problems
- make state-of-the-art implementations of MP solvers accessible and provide a unified interface to them

– provide the means to safely combine standard MP optimisation solvers and propagation-based CP solvers

While the first two objectives give rise to relatively straightforward engineering tasks, everyone who wants to smoothly integrate CP and MP faces the dilemma that the standard solution techniques exhibit quite fundamental differences:

In CP, problems are solved by a combination of propagation (the systematic exclusion of non-solutions from the search space) and search (the heuristic partitioning of the search space into smaller, more manageable sub-spaces). This principle is very general and not specific to a particular problem class. It is however aimed at constraint satisfaction, i.e. at finding all solutions that satisfy the constraints. If an objective function is given, and optimisation is required, then this is usually achieved by applying a bounding method on top of the all-solutions method, i.e. incrementally looking for solutions that are better than a previously found one.

In MP, we deal with two particular problem classes: linear programming problems (LP, linear constraints over continuous variables), and (mixed) integer problems (MIP, where some or all of the variables are required to take integral values).

For LP, we have very good algorithms (simplex and interior point methods) which, given a constraint system and an additional objective, can find one (of possibly many) optimal solutions quite efficiently. But unlike constraint propagation techniques, these algorithms do not compute a representation of a reduced search space, and can therefore not straightforwardly be integrated into a CP system.

MIP solution techniques are also difficult to integrate with CP, but for a different reason. Like for general CP, there is no efficient direct algorithm for solving MIPs. MIPs are therefore solved by a combination of branch-and-bound search with an underlying LP solver. When integrating with CP, we face the problem of having to merge the CP and the MIP search. Because the MIP search is usually implemented as a black box, this is difficult to achieve.

3 Functionality

Like other solvers in ECL^{i}PS^e, *eplex* is implemented in the form of a library. The *eplex* library allows MP problems to be modelled in ECL^{i}PS^e, and solved (optimised) by an external MP solver. In terms of solving, we provide two interfaces, a low-level procedural one that is close to the MP solver's given API, and a safe, logical one that is close to the concepts used in propagation-based CP solvers. In either case, our interfaces try to hide as much as possible the differences between the different brands of MP solvers that we interface to.

3.1 Declarative Modelling

Problems are modelled in the same way as with other ECL^{i}PS^e solvers, i.e. by specifying a logic program where some of the predicates represent the constraints to be satisfied. The constraints that *eplex* supports are equalities and inequalities (`$=/2`, `$>=/2`, `$=</2`) over linear arithmetic expressions, the integrality constraint (`integers/1`) and the bounds constraint (`$::/2`) which is just a special case of inequalities. They are used as in the following examples:

```
:- lib(eplex).

model1([X,Y], X) :-
    X $= Y + 1,
    integers([X]),
    X + Y $>= 4.

model_knapsack(Bools, Weights, Profits, Cap, Profit) :-
    Bools $:: 0..1,
    integers(Bools),
    Bools*Weights $=< Cap,   % List*List (dot product)
    Bools*Profits $= Profit.
```

The first example, `model1`, defines a problem with two variables, one equality, one inequality and one integrality constraint. The second example, `model_knapsack`, is a model for a knapsack problem, consisting of the declaration of a list of boolean variables, and stating a capacity constraint and a profit equation, using the given lists of item weights and profits.

Note that this modelling code is solver-independent, in particular, it is completely identical to the modelling code that would be used for ECL^{i}PS^e's interval and finite-domain propagation solver *ic*!

3.2 Procedural Solver Interface

Once a problem has been modelled by means of constraints, we want to solve it. We first present a conventional, procedural interface that is very close to the API provided by the MP solvers, and not too different from the way an imperative language would interface to such a solver. In addition to the constraint predicates that are used in the problem model, we simply provide additional primitives for (i) specifying the objective, (ii) invoking the solver, and (iii) accessing solution information. The model above can then be solved using this additional code:

```
solve :-
    model1([X,Y], Cost),
    eplex_solver_setup(min(Cost)),
    eplex_solve(Opt),
    eplex_var_get(X, solution, OptX),
    eplex_var_get(Y, solution, OptY),
    printf("Solution X=%f, Y=%f at cost %f%n", [OptX,OptY,Opt]).
```

Setup and solving Solver setup is performed via the `eplex_solver_setup/1`, which initialises the MP solver for this problem. Constraints can be stated before or after solver setup. In this case, we stated the constraints before solver setup. Operationally, constraints stated before solver setup are delayed, and during solver setup, the objective is set, and all constraints stated so far are collected and passed to the solver in one batch. No solving is performed at this time: this is the job of the separate `eplex_solve/1` predicate, which invokes the MP solver and either fails (if the problem is infeasible) or returns the optimal objective value. The point of separating setup and solving functionalities is to enable repeated re-solving of the problem, usually with a slightly modified

problem. After the initial problem setup, additional constraints and variables can be incrementally added to the problem (and indeed be removed on backtracking) in the same way this would happen with the constraints of a CP solver. Problem modification can be interleaved with new calls to the solver.

Accessing results Apart from computing the optimal objective, the solver also finds one set of solution values for the problem variables. These values must be explicitly retrieved using the `eplex_var_get/3` predicate. This may seem surprising at first - why are the variables not instantiated to their solution values? There are two reasons: First, the MP solver delivers only one of possibly many solutions, so it would not be logically correct to assign this value. In particular, this initial solution may no longer be feasible once additional constraints are added and the problem is re-solved.

A second problem is that MP solvers are typically implemented using floating-point arithmetic, making the results subject to rounding errors. This means that with non-integral variables, the floating-point values that the solver considers a solution will usually not be suitable to serve as actual variable instantiations. In particular when equality constraints are involved, passive checking of the constraints with floating-point solutions filled in will usually fail (because we are violating the golden programmer's rule of never comparing floating-point values for equality). All non-integral solutions from an MP solver should therefore always be considered as approximations. A good use for them is as a labelling heuristics within a search routine.

The `eplex_var_get/3` predicate is also used to retrieve other variable-related solver information, like the reduced costs, which are useful to do cost-based filtering [12]. Similar interface predicates give access to further information from the solver, for instance dual values for the constraints.

3.3 Logical Integration with CLP

The problem with the procedural interface described above is that the semantics of the posted constraints is only respected when the programmer explicitly invokes the `eplex_solve/1` predicate in the right places of the code.

The programming paradigm of a CP system is however that the constraints 'take care of themselves', i.e. once they have been posted, the system should automatically make sure that they are not violated. And not only that: much of the power of CP derives from the data-driven way in which the consequences of changes are propagated through a constraint network. Each CP constraint is represented by one or more propagators, which are suspended awaiting specific events (which generally involve changes to the variables in the constraint) that will trigger their execution.

We now combine the availability of the MP solver with the idea of event-driven execution in order to achieve a logically correct implementation of our linear constraints, to detect inconsistency as soon as possible, and even to propagate information in case of consistency. We just need to make sure that the solver is automatically invoked whenever the corresponding constraint system has changed, more precisely, if it was tightened in a way that invalidates the previously found solution. This may happen through

- the addition of new linear constraints (`new_constraint`)

- new, tighter variable bounds that exclude the solution value (`deviating_bounds`)
- instantiation of variables to a value different from its solution value (`deviating_inst`)

The *eplex* library supports all these (and a few more) trigger conditions, which can be specified as parameters to an extended version of the `eplex_solver_setup` predicate. Note that the laziest trigger condition that still achieves logically correct handling of the constraints is the `deviating_inst`-condition: when variables are instantiated to a non-solution value, the solver is reinvoked. In particular, it is guaranteed that the constraints will have been checked (and inconsistency detected) once all variables are instantiated. Additional trigger conditions will make the system more eager: when using all three conditions above, inconsistency will be detected as soon as possible.

Additionally, the programmer can specify their own triggering conditions based on the general suspend and resume mechanism of ECL^{i}PS^e. The MP solver can thus be made to trigger only on 'interesting' problem changes, reducing unnecessary computation in case the predefined trigger conditions turn out to trigger too eagerly.

An automatically triggered solver can do more than just detecting inconsistency: it computes an optimum cost for the current state of the constraint system. Since the constraints can only get tighter later, this cost can be used as a lower bound (in case of minimisation) on the cost variable (C in the example code). The MP solver thus acquires the characteristics of a propagation constraint: it reacts to e.g. bound changes in its problem variables, and imposes a new bound on its cost variable. It can therefore take part in propagation sequences, and it can be considered as a compound constraint, representing the whole MP problem with all its variables.

In a setting where (cheap) interval-propagation constraints are mixed with (expensive) MP-solver constraints, we prioritise the execution such that the expensive constraints are only executed once the cheap constraints have reached a fixpoint. That way, MP solving is done only as many times as absolutely necessary.

Another way in which a solver can perform propagation is by pruning variable bounds using reduced cost information [12]. This feature is available as a further solver setup option. Other information from the MP solve can be used to assist the CP solve more indirectly, e.g. using the solution values for labelling the variables.

3.4 Multiple Subproblems

Since the constraint syntax is identical for different solvers, in a hybrid program it becomes necessary to specify which solver a constraint is intended for. This is syntactically solved by prefixing the constraint with the solver name (which is in fact simply an ECL^{i}PS^e module name), e.g. with `ic:(X+Y $>= 4)` the constraint is posted to the *ic* interval CP solver, with `eplex:(X+Y $>= 4)` the constraint is posted to the *eplex* MP solver.

We also wanted to provide the flexibility to group *eplex* constraints into separate subproblems that can then be handled as independent subproblems by an MP solver. This is done through the concept of *eplex* instances. The constraints are prefixed with instance names to group them into different subproblems. The following example defines two overlapping subproblems, corresponding to the two declared *eplex* instances 'lp' and 'mip':

```
:- eplex_instance(mip).
:- eplex_instance(lp).

model2([X,Y,Z]) :-
    [lp,mip]: (X $= Y + 1),
    mip: integers([X]),
    [lp,mip]: (X + Y $>= 4).
```

Each constraint here is posted to one or both solver instances. Note that not only constraints, but also the solver setup, solve and access predicates can be prefixed by an instance name in order to make them apply to a particular instance. The solver-prefix is first-class, i.e. it can be a variable and specified at runtime.

3.5 Branch-and-cut

Using the automatic triggering mechanism, the solving of an *eplex* problem can be tightly integrated into the CP system's constraint propagation and search process. The most natural example of this is the implementation of a branch-and-cut search. Here, some of the constraints of the full problem are initially relaxed. Branching is then done by adding different constraints to the problem on each branch of a search node.

The simplest example of this in MP is the MIP search. It is used to obtain the optimal integral solution to a problem, where at each node, a relaxed linear problem is solved, and in each branch constraints are added to push integer variables away from non-integer solution values. This search can be implemented in ECL^{i}PS^e, using *eplex* to solve the relaxed problem automatically when required:

```
example_mip(Vars, Opt) :-
    model(Vars, Ints, Obj),                  % problem specification
    eplex_solver_setup(min(Obj), Opt, [], [deviating_bounds]),
    bb_min((branch(Ints),eplex_get(cost,Opt)), Opt, _).        % (A)

branch(Ints) :-
    (
        member(X, Ints),                                       %
        eplex_var_get(X, solution, Sol),                       % (B)
        abs(Sol - round(Sol)) >= 1e-5                          %
    ->
         ( X $=< floor(Sol) ; X $>= ceiling(Sol) ),            % (C)
         branch(Ints)
    ;
         true                                % integer solution found
    ).
```

For this program, we are using two features of ECL^{i}PS^e to perform the required search: Disjunctions `';'` with automatic depth-first search to explore all alternatives (line C), and the generic branch-and-bound control procedure `bb_min` to impose cost bounds and locate an optimal solution (line A).

We use the MP solver only to solve the continuous relaxation of the problem, and take care of integrality constraints explicitly. At each search node, we select a variable whose value should be integral, but is indeed fractional in the current solution to

the continuous relaxation (lines B). The program then branches by adding (bounds-) constraints to the problem to push the solution away from the non-integral value (line C).

The *eplex* problem is set up with a call to `eplex_solver_setup/4`, which allows the user to customise the solver setup. The last argument specifies the pre-built `deviating_bounds` trigger condition, so the solver is triggered by the exact condition we need to push the solution value of an integer value away from its fractional value: if it is fractional and is outside the new bound, the solver would be invoked. By default, if trigger conditions are specified, then the problem will be solved once immediately after set up, so that there is a solution available when the solver is triggered. Also, in general, if a branching decision does not affect the MP problem variables (as specified by the trigger condition), then the solver is not invoked.

The above example is a very simple implementation of a MIP search and is of course not competitive with the MIP search built into the MP solver. However, this search framework can be used to implement more flexible and elaborate search strategies that cannot be performed by the black-box MP solver alone. Indeed the subproblem solved at each node can be a MIP or any of the problem types supported by the MP solver. Examples of this more involved search are probe backtrack search [9] and its generalisation [2], which was used to implement a commercial transportation application.

4 Implementation Considerations

4.1 Outline of Implementation

The *eplex* library is written in both ECL^{i}PS^e and C, corresponding to the logical and low-level interfaces outlined in section 3. Two MP solvers are currently supported: Dash Optimization's Xpress-MP [15], and ILOG CPLEX [17]. Because of the differences between the two solvers' API and because not all *eplex* features are directly supported by both solvers, the C layer contains some solver dependent code. The ECL^{i}PS^e layer is almost completely solver independent.

Each eplex instance is implemented as a problem instance of the MP solver. The problem instance is created when `eplex_solver_setup` is called, using the objective function and any constraints that have been posted to the eplex instance. In this phase, the main job of the eplex interface is to convert the ECL^{i}PS^e modelling level representation of the problem's variables and constraints into the form required by the MP solver, namely a compact row- or column-wise matrix representation. Constraints posted *after* initial solver setup are added to the problem instance incrementally.

In terms of data structures, each *eplex* problem is represented by a problem handle at the ECL^{i}PS^e level. This is simply a Prolog structure storing a reference to the MP solver instance plus various information associated with the problem, e.g. the solution values for the variables. The ECL^{i}PS^e level variables are linked to the solver by means of variable attributes (now a feature of several popular Prolog implementations): each problem variable is given an *eplex* attribute which refers to the problem handle. If a variable occurs in more than one eplex instance, a chain of *eplex* attributes is created, one for each eplex instance.

The data-driven triggering of the MP solver is implemented using the suspension (delayed goal) mechanism of ECL^{i}PS^e. A 'demon' goal which invokes the MP solver is created, which is woken and executed whenever the specified triggering conditions are met.

Any changes made to a problem after setup (e.g. adding constraints, changing variable bounds), need to be undone on backtracking to maintain the 'logical behaviour' of the whole system. As far as ECL^{i}PS^e level data structures are concerned, this undoing is automatic. The challenge for the *eplex* interface is to make changes in the external MP solver behave in the same way. This is done at the C level with ECL^{i}PS^e's 'trail undo' facility, which allows a C function call to be trailed on forward execution, and executed when it is untrailed. Several changes are undone this way, the most important is to restore the original problem matrix after backtracking. As constraints posted after problem setup are appended to the end of the matrix, the original matrix is restored simply by resetting the matrix to its former size. It should be noted that for some types of incremental changes, the use of a time-stamping technique[1] is essential in order to avoid excessive trailing.

None of the features required to implement the *eplex* library are specific to ECL^{i}PS^e: an interface to C/C++, suspension, attributed variables, and a 'trail undo' facility, are supported by other CLP systems. It should thus be possible to implement the *eplex* library in other CLP languages, although not trivial due to lack of standardisation.

4.2 Overheads of Performing Search in ECL^{i}PS^e

A main concern is the efficiency of the common scenario of conducting search in ECL^{i}PS^e while solving multiple subproblems. Can a high-level language like ECL^{i}PS^e efficiently maintain the search-tree needed and can it allow an MP problem to be efficiently modified and solved repeatedly?

An issue is how the successive subproblems are produced. In many cases, the successive subproblems are derived from each other with small changes, and *eplex* will allow the same MP problem to be incrementally changed and re-solved. This should be more efficient than the alternative, which is to construct each subproblem afresh for each solve.

We tried to measure the impact of incremental modifications and maintaining the search-tree in ECL^{i}PS^e by timing various ways of performing MIP, the most common MP search method. Firstly, we perform the MIP search using the MP solver. Secondly, we perform the MIP search in ECL^{i}PS^e, using the MP solver as a linear solver at each node, and allowing the problem to be incrementally modified. Thirdly, we perform the MIP search in ECL^{i}PS^e as before, but construct the problem afresh at each node.

The MIP problem used for this study is taken from a set of examples that originated from MIPLIB [4], a standard MIP benchmark suite.

The results, obtained on a 900MHz Pentium III Linux box with 256M of memory, running ECL^{i}PS^e 5.8 with CPLEX 8.1.1, are presented in Figure 1. For each problem, its size in terms of number of variables (vars) and constraints (cons) are given. For most of the problems with the solver MIP and incremental MIP, the solving is repeated 10 to 100 times to get a more accurate timing, and each timing is done 3 times.

Program			CPLEX mip		ECLiPSe incr. mip		ECLiPSe non-incr mip		
	vars	cons	nodes	node^{-1}	calls	node^{-1}	nodes	node^{-1}	load
flugpl	18	17	70	0.393	4957	0.339	5221	3.33	1.74
flugplan	18	17	70	0.404	1986	0.393	2187	3.22	1.65
sample2	67	45	75	1.79	353	0.652	345	6.58	4.47
noswot	128	182	1	55.1	1127	1.63	32786	17.0	13.3
bell3a-nonred	133	111	18845	1.19	142583	1.45	162027	14.2	9.44

Fig. 1. Performing MIP search in ECL^{i}PS^e

In the table, we give either the number of nodes in the MIP search tree (including the root node), or (in the incremental case, where there is one solver call per MIP node), the number of solver calls, and the derived average time spent per node (runtime divided by the number of nodes) of the search tree. In addition, the last column in the table is the time needed to load the initial problem once into the MP solver, all timings are in ms. The problem is constructed and loaded 1000 times, to simulate the construction of the problem in the non-incremental MIP search.

MIP search-tree size A simple depth-first branch-and-bound search similar to that outlined in section 3.5 was used for the ECL^{i}PS^e MIP. The MP MIP search benefits from good branching decisions and other optimisations, and its MIP search-trees are significantly smaller than ECL^{i}PS^e's. Our interest in this study is not how good the MP's MIP strategy is, but in the overheads in performing a search in ECL^{i}PS^e. For this, the time spent on each node of the search-tree is a more accurate reflection of the overheads associated with implementing the search.

Incremental vs. non-incremental search Even for small problems like the ones tested, the ECL^{i}PS^e incremental MIP search is about 10 times faster per node than the non-incremental version. Modifying an existing problem is much less costly than constructing the problem anew, as loading the problem is relatively expensive. In addition, the incremental case is able to 'warm start' a problem when the modified problem is re-solved – the solver will not start solving from scratch, but instead will try to reuse information from the previous solve.

ECL^{i}PS^e search vs. MP search Comparing the incremental search with the MP solver's MIP search is somewhat more complicated: the MP's MIP search is tightly integrated with its linear solver, and this should result in lower overheads in the solving of each node, for example, adding the constraints at each node can be done more directly. Furthermore, some optimisations can be done once at the start of the MP solver's MIP search, rather than repeatedly at each node, as is the case for the ECL^{i}PS^e MIP search. At each node, the MP MIP search can also take advantage of the knowledge that it is performing a MIP search, for example by posting extra cuts that would be invalid for the LP problem and the problem may even be solved more than once per node to drive it closer to an integer solution. However, the effect of this may be more to reduce the search-tree size, rather than make the solve at each node faster: the `noswot` result is a striking example of this: the MP MIP solves the problem with a single node, but this

solve itself is relatively expensive, even taking into account the cost of loading the problem. Thus, the time per node comparison presented in the results should be taken with care.

As the size of the MIP search-tree is so small for many of the MP MIP search, it is not too meaningful to use the per node time for comparison: the cost of loading the problem into the solver, and the cost of performing the initial solve, which is likely to be more expensive because it does not benefit from a warm start, will skew the results too much. However, for `bell3a-nonred`, where the search-tree is sufficiently large, the time per node for the MP MIP and incremental MIP are quite similar, suggesting that the cost of using ECL^{i}PS^e to control the MIP search is not prohibitive.

Impact on real applications This ability to solve multiple problems, and to repeatedly modify and solve problems has been used to good effect to solve very large problems. In some applications, over a million subproblems were solved in a single program, e.g. [7], which performs a complex search and at each node solves a series of subproblems, including some that have a quadratic objective.

4.3 Memory Considerations

Multiple representation of problem For large problems, the memory required to represent the MP problem can be significant. Moreover, the problem may be represented in different forms at the same time during the execution. At the ECL^{i}PS^e level, the constraints for the problem are initially represented as expressions. When they are added to the MP solver, they are first converted to a normalised form, and then passed to the C level to construct the data structures required by the MP solver API. Both the C data structures and the normalised form are only required temporarily, and the memory used can be recovered once the constraints have been passed to the MP solver.

If a constraint is required by the MP solver and another ECL^{i}PS^e solver, then it has to be represented in both. If it is only required by the MP solver, then the ECL^{i}PS^e representation can be dropped once it has been passed to the MP solver. This is done automatically by ECL^{i}PS^e if the constraint is posted incrementally to the MP solver and then not referred to elsewhere in the program.

For most applications, the constraints for *eplex* are not given statically in the model code, but are computed from some sort of abstract representation of the problem, e.g. a graph. This will impose extra memory usage on the program.

MP representation of the problem The MP solver stores the problem in a compact form, with only the non-zero coefficients of the constraints stored (along with their location in the problem matrix). In ECL^{i}PS^e, the constraint are represented as expressions, which also normally contain only non-zero coefficients. However, more memory is required to store the expression as it is designed for ease of manipulation rather than minimise memory usage. The exact amount of memory required depends on the actual expressions used, but is roughly about 4 to 5 times greater than that of the compact form.

A concrete example We examined Thorsten Winterer's Swapper program from his thesis [24], which is an application to swap aircrafts for scheduled flights. We examine his single MIP formulation of the problem, and the largest problem instance he used: this extracted a MIP problem from a graph constructed from the raw flight data, and has 421473 constraints and 145278 variables. As written, the program first constructs all the constraints, before posting them all to the MP solver in one go.

However, as the constraints are not used elsewhere at the ECL^{i}PS^e level, and most of the constraints can be extracted without looking at the whole graph, they can be posted to the MP solver immediately. We modified the program to do this, and the peak memory usage was greatly reduced: from about 400M to 150M. The execution time (to the point where the problem have been loaded into the MP solver), however, increased slightly from 102 seconds to 126 seconds (on a Pentium 4 2GHz Linux box with 1G of memory, running CPLEX 9.0). This is probably due to the increase in memory management in the solver when the constraints are added to it incrementally.

In summary, while the ECL^{i}PS^e representation of the problem is less compact than the compact matrix representation, it is often not necessary to represent the whole problem at the ECL^{i}PS^e level. In addition, even though the ECL^{i}PS^e representation is less compact, it still avoids representing the non-zero coefficients, and would use far less memory than a full, non-sparse matrix for the problem, so it is still possible to represent quite large problems. At IC-Parc, *eplex* has been used successfully to solve problems that approach 1 million constraints and variables e.g. [25] (627168 variables, 947967 constraints).

5 Related Work

5.1 Extensions of *eplex*

In addition to direct use of *eplex* in applications, *eplex* is also used at IC-Parc to develop various hybridisation forms, such as column generation [10], Bender's decomposition [11] and Lagrangian relaxation [20]. Of these, column generation is now packaged as an ECL^{i}PS^e library. In addition to the facilities described in this paper, *eplex* provides additional low-level support for the *colgen* library, for example, adding new columns with non-zero coefficients in existing rows of the matrix.

5.2 CLP Systems that Perform MP Solving

An alternative to providing an interface to an external MP solver is to implement an MP solver. In this case, it should be possible to achieve much tighter coupling between the MP solving and the rest of the CLP system, e.g., there may be no need to construct a separate representation of the problem for the MP solver as in *eplex*. In fact, this is the approach taken by many of the earlier CLP systems that have constraint solvers over the real domain, such as CLP($\mathcal{R}$) [18], and clp(Q,R) [16], both of which implemented their own Simplex solvers.

For CLP($\mathcal{R}$), the solver is used to determine the feasibility of a set of constraints, rather than finding an optimal. The tighter integration of the solver with the rest of the

CLP system allows the posted constraints to be actively simplified as new constraints are added. However, this ability to rewrite constraints means more complex backtracking actions are required to restore the constraints: unlike in *eplex*, where the problem matrix can simply be restored to its original size on backtracking. The Q variant of clp(Q,R) performs all calculations with rational rather than floating point values, avoiding imprecision problems at the cost of increased time and memory. Another difference with *eplex* is that both CLP($\mathcal{R}$) and clp(Q,R) do not provide mechanisms for separating the constraints into different subproblems that are solved independently.

Performance comparison A motivation for the *eplex* interface is that an external MP solver would be more efficient than trying to implement an MP solver directly – considerable effort and specialist knowledge have been devoted to implementing MP solvers such as CPLEX and Xpress, and it is unlikely that similar effort (and indeed the specialist knowledge) can be devoted to a single component in a CLP system. To see if this belief is correct, we compared *eplex* using both the CPLEX and Xpress MP solvers against clp(Q,R). clp(Q,R) has a Simplex solver implemented in Prolog with attributed variables, and can optimise LP and MIP problems. It is available with several CLP systems, including ECL^{i}PS^e. Some effort was spent to implement an efficient Simplex solver, although the MIP search implementation is still quite a simple one.

Program	clpr,eclipse		eplex,CPLEX		eplex,xpress	
	lp	mip	lp	mip	lp	mip
flugpl	0.0087s	2.43s	1.43×	88.4×	1.21×	33.4×
flugplan	0.0089s	0.99s	1.56×	35.0×	1.27×	13.5×
sample2	0.17s	4.31s	12.7×	32.2×	12.7×	29.3×
noswot	2.95s	–	78.5×	(0.0551s)	67.8×	(2.31s)
bell3a-nonred	4.53s	20472s	164×	913×	168×	957×

Fig. 2. Speedup comparison of clp(Q,R) with eplex

Figure 2 shows the speedups of solving the same problems used in the search comparison (section 4.2) with *eplex* (using CPLEX 8.1.1 and Xpress MP 14.27), relative to the performance of the R solver of clp(Q,R). The results were obtained using a 900MHz Pentium III Linux box running ECL^{i}PS^e 5.8. The problems are solved as both LP (where the integer constraints are dropped) and MIP problems, and the timings are presented for clp(Q,R) (in seconds) and for the MIP `noswot` times for CPLEX and Xpress, as the clp(Q,R) was unable to solve this problem due to stack overflow.

Except for the smallest problems (`flugpl` and `flugplan`), *eplex* with the two MP solvers was significantly faster than clp(Q,R) running on ECL^{i}PS^e: between 1 and 2 orders of magnitudes for the linear problems, and 1 and 3 orders of magnitudes for the MIP problems.[1] In addition, the difference is greater for the larger problems, so the

[1] The performance of the ECL^{i}PS^e version of clp(Q,R) is quite comparable to that on other CLP systems. For example, the measured difference in execution time between ECL^{i}PS^e and SICStus Prolog (version 3.11.2) running these problems is at most 25%.

difference would likely even be greater for the type of application problems that have been tackled using *eplex*.

In addition to the performance advantages, the external MP solvers offer more options. For example, both Primal and Dual Simplex and interior point methods are available for solving problems, and quadratic problems (i.e. problems with quadratic objectives) can be solved.

5.3 Other Ways of Combining CP and MP

Using problem files Instead of interfacing to the callable library of the MP solver, an alternative would be to generate a file specifying the problem in one of the standard formats (MPS or LP) that can then be read in and solved by an MP solver. The solution is written to a file and read back by the user program. This provides a looser coupling between the CLP language and the MP solver, and is probably easier to implement for solving individual MP problems. This approach was used initially to interface ECL^{i}PS^e to an MP solver [14], before the development of *eplex*, and was also used by COSYTEC to combine CHIP [3] with Xpress MP to solve a part of a train schedule problem.[2] However, it is less flexible than using the callable library. For example, it would be difficult to repeatedly modify and resolve the same problem, without creating the problem anew each time, and it would be difficult to achieve tight co-operation between the MP solver and other solvers in the CLP system.

Other high-level languages combining MP and CP *Eplex* allows MP and CP problems to be modelled in a high-level language. In the MP community, the need for a easy-to-use way of modelling MP problem lead to the development of modelling languages such as AMPL [13] and GAMS [5]. This in turn lead to the development of OPL [22], which extended MP modelling languages to model and solve CP problems as well. However, like other MP modelling languages, OPL lacks the flexibility of a full-blown programming language, and to allow a problem to be decomposed into subproblems that are solved separately, a scripting language, OPL Script [23], was introduced. As each subproblem can be solved by different methods, it does allow some hybrid solving. Additionally, limited predefined ways of combining CP and MP solving in the same OPL model is also possible, but as OPL Script is separate from the OPL model, more programmatic control is not possible within the search specification. In addition, the only way available to modify a problem and re-solve it is to change the data (constraints) associated with the OPL model using OPL script, and then re-initialise the model. This appears to create a new instance of the problem, which can be much more expensive than incremental changes of the problem, as discussed in section 4.2.

Although OPL/OPL Script is solver independent, it is currently available with ILOG CPLEX and Solver only.

Xpress-Mosel [8] offers high-level language functionality with Xpress-MP, and with the announcement of the constraint-base module Xpress-CP, which uses the constraint engine of CHIP, similar functionality to OPL is available.

[2] Personal communication with Helmut Simonis, 2004.

Combining MP and CP in an imperative language Using a high-level language to combine CP and MP is not the only possibility. Much existing hybrid research work is done using C/C++, interfacing to MP solvers such as ILOG CPLEX and CP solvers such as ILOG Solver. In fact, ILOG provides Concert Technology [17], a common C++ classes and functions for Solver and CPLEX, to aid the writing of such code. With this approach, the user would not benefit from the high-level ease of programming provided by a language such as ECL^{i}PS^e or OPL, and furthermore, the programs are no longer solver independent.

5.4 Other Common Solver Interfaces

The MP modelling languages like GAMS and AMPL are both available for use with different MP solvers. Unlike OPL, however, they do not provide a CP solving component.

The Open Solver Interface (OSI) from the COIN-OR project [6, 21] is a uniform API in C++ for calling MP solvers. This allows solver independent program code to be written in C++. In 2004, we investigated if *eplex* can use OSI as the API, rather than directly the CPLEX and Xpress API, for accessing the MP solvers. This would also give *eplex* immediate access to other solvers such as GNU's GLPK. A prototype *eplex*-like interface, implementing the minimal required functionality, was developed. However, at the time, the OSI API was not flexible enough, particularly for MIP problems, to replace our existing interface.

6 Conclusion

We believe that *eplex* provides a very powerful and flexible interface for users to solve problems with MP and hybridisation techniques within a CLP language. While it is now implemented for ECL^{i}PS^e, it should be possible to adapt it for other CLP languages.

The interface is still evolving to meet the needs of our users. In the short term, we plan to add support for globally valid constraint pools ('global cuts' pools) – once added to the pool, these constraints will apply to all subsequent solving of the problem, even after backtracking.

We also plan to support Bender's Decomposition and Lagrangian Relaxation as libraries for ECL^{i}PS^e, so that the techniques can be used by the general users without reprogramming these techniques on their own.

Acknowledgements

The authors gratefully acknowledge the invaluable help we got from our colleagues at IC-Parc, in particular Andy Eremin, through their feedback and discussions on the development of eplex. We also thank Andy Eremin and Andy Cheadle for their comments on drafts of this paper. Many thanks also to Roland Yap for providing us with CLP($\mathcal{R}$), and answering our questions about it. Finally, we would like to thank Dash Optimization, for their help and support with our use of the Xpress MP solver.

References

[1] A. Aggoun and N. Beldiceanu. Time Stamps Techniques for the Trailed Data in Constraint Logic Programming Systems. In *Actes du Séminaire 1990 - Programmation en Logique*, 1990.

[2] F. Ajili and H. El Sakkout. A Probe-based Algorithm for Piecewise Linear Optimization in Scheduling. *Annuals of Operations Research*, 118, 2003.

[3] N. Beldiceanu, H. Simonis, P. Kay, and P. Chan. The CHIP System. White Paper COSY/WHITE/002, COSYTEC SA, 1997.

[4] R. E. Bixby, C. M. M. S. Ceria, and M. W. P. Savelsbergh. An Updated Mixed Integer Programing Library: MIPLIB 3.0. Technical Report TR98-03, The Department of Computational and Applied Mathematics, Rice University, 1998.

[5] A. Brooke, D. Kendrick, A. Meeraix, and R. Raman. *GAMS A User's Guide*, 1998.

[6] COIN-OR Foundation. COIN-OR Website http://www.coin-or.org.

[7] W. Cronholm and F. Ajili. Strong Cost-Based Filtering for Lagrange Decomposition Applied to Network Design. In *CP 2004*, 2004.

[8] Dash Optimization. *Xpress-Mosel User Guide*, 2004.

[9] H. El Sakkout and M. G. Wallace. Probe Backtrack Search for Minimal Perturbation in Dynamic Scheduling. *Constraints*, 5(4):359–388, 2000.

[10] A. Eremin. *Using Dual Values to Integrate Row and Column Generation into Constraint Logic Programming*. PhD thesis, IC-Parc, Imperial College London, 2003.

[11] A. Eremin and M. Wallace. Hybrid Benders Decomposition Algorithms in Constraint Logic Programming. In *CP 2001*, 2001.

[12] F. Focacci, A. Lodi, and M. Milano. Cost-based Domain Filtering. In *CP 1999*, 1999.

[13] R. Fourer, D. M. Gay, and B. W. Kernighan. A Modeling Language for Mathematical Programming. *Management Science*, 36, 1990.

[14] M. T. Hajian, H. El-Sakkout, M. Wallace, J. M. Lever, and E. B. Richards. Towards a closer integration of finite domain propagation and simplex-based algorithms. *Annals of Operations Research*, 1998.

[15] S. Heipcke. *Applications of Optimization with Xpress^{MP}*. DASH Optimization Ltd., 2002. Translated and revised from the French Language.

[16] C. Holzbaur. Ofai clpq(q,r) manual, edition 1.3.3. Technical Report TR-95-09, Austrian Research Institute for Arificial Intelligence, Vienna, 1995.

[17] ILOG, Inc. ILOG Products Web Page: `http://www.ilog.com/products/`.

[18] J. Jaffar, S. Michaylov, P. Stucky, and R. Yap. The CLP(R) Language and System. *ACM Transaction on Programming Language Systems*, 14(3), 1992.

[19] M. Milano, editor. *Constraint and Integer Programming: Toward a Unified Methodology*. Kluwer Academic Publishers, 2004.

[20] W. Ouaja Ajili. *Integrating Lagrangian Relaxation and Constraint Programming for Multicommodity Network Routing*. PhD thesis, IC-Parc, Imperial College London, 2004.

[21] T. Ralphs. COIN-OR: Software Tools for Optimization. Tutorial at CORS/INFORMS Joint International Meeting, May 2004.

[22] P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.

[23] P. Van Hentenryck and L. Michel. OPL Script: Composing and Controlling Methods. In *New Trends in Constraints*, 2000.

[24] T. J. Winterer. *Requested Resource Reallocation with Retiming: An Algorithm for Finding Non-Dominated Solutions with Minimal Changes*. PhD thesis, IC-Parc, Imperial College London, 2004.

[25] Q. Xia. Traffic Diversion Problem: Reformulation and New Solutions. In *Proceedings of the Second International Network Optimization Conference*, 2005.