A presentation of FLOPC++

Tim Helge Hultberg
Critical Software SA /University of Aveiro
email: thh@mat.ua.pt

July 17, 2003

Abstract

FLOPC++ is an open source algebraic modelling language, implemented as a C++ class library, which can be downloaded at `www.mat.ua.pt/thh/flopc`. Using FLOPC++, linear optimization models can be specified in a declarative style, similar to algebraic modelling languages such as GAMS and AMPL, within a C++ program.

1 Introduction

Real world optimization problems typically arise as specific data instances of generic algebraic models, while the software used to solve these problems (the solver) expects the problems to be passed to it in an explicit matrix format (typically with a sparse column compressed storage of the coefficient matrix). The benefits of using algebraic modelling languages to automate the generation of problem instances are well recognized and a number of algebraic modelling languages have emerged. However, traditional algebraic modelling languages have shortcomings with respect to:

Embedding of optimization models in applications. The data entering an optimization model typically comes from sources such as databases, spreadsheets and GUI's and its solution might be needed as input for all kinds of further processing. An optimization model is often just a minor part of a larger software application, but this embedding is hard to achieve with traditional algebraic modelling languages.

Implementation of model tailored solution algorithms. The algebraic description of a model convey structural information that is hidden in the matrix representation required by the solver. Sometimes this information can be used to develop efficient specialized solution algorithms while still depending on a general linear optimization solver. Examples of such algorithms include decomposition, column generation and model specific cutting plane algorithms. The development of such algorithms is greatly facilitated by working in an environment where the algebraic description of the model is available. But efficiency considerations make traditional algebraic modelling languages a bad choice.

These shortcomings can be remedied to a certain extent by incorporating procedural features and links to external software (such as databases and spreadsheets) into the languages. But bringing modelling features to a programming language (rather than the opposite) is a much more natural and versatile approach.

A small step in this direction has been taken with the ILOG Concert Technology, which provides support for mapping of variable/constraint indices to instance column/row indices, but offers only an executable (as opposed to declarative) model representation. FLOPC++, on the contrary, takes the full step by making declarative model representation (similar to traditional algebraic modelling languages) possible within C++ programs.

2 Optimization modelling and C++

Optimization modelling and C++ is a perfect match. Due to the power of object-oriented programming and the standard template library (STL), an efficient algebraic modelling language, including sparse multi-dimensional subsets and index arithmetic, can be implemented as a C++ class library using less than 3000 lines of code. Linear optimization models can then be conveniently specified within a C++ program using modelling objects (sets, data, variables and constraints) declared as instances of classes provided by FLOPC++, making it easy to integrate such models in software applications.

An example of a FLOPC++ representation of a sparse transportation model is shown below:

```
class Transport : public MP_model {
    MP_data COST;
    MP_variable x;
    MP_constraint supply, demand;
public:
    Transport(MP_set& S,          // Sources
              MP_set& D,          // Destinations
              MP_subset<2>& Link, // Transport. links (sparse subset of S*D)
              MP_data& SUPPLY, MP_data& DEMAND, MP_data& DISTANCE) :
        MP_model(new OsiOslSolverInterface),
        COST(Link), x(Link), supply(S), demand(D) {

        // Assignment of derived data
        COST(Link) = 90 * DISTANCE(Link) / 1000.0;

        // Definition of constraints
        supply(S) =  sum( Link(S,D), x(Link) ) <= SUPPLY(S);
        demand(D) =  sum( Link(S,D), x(Link) ) >= DEMAND(D);

        // Objective function
        minimize( sum(Link, COST(Link)*x(Link)) );
    }
};
```

The use of a class (derived from `MP_model`) to specify the model is not required, but can be advantageous when several models are used. (A number of complete example models are available at the FLOPC++ Home page: `www.mat.ua.pt/thh/flopc`.)

The declarative representation is possible because expressions in FLOPC++ evaluate to abstract syntax trees, which can subsequently be analyzed and used for generating problem instances. Reference counting is used to avoid memory leaks due to the dynamic allocation of the data-structures representing the abstract syntax trees.

FLOPC++ uses the COIN (`www.coin-or.org`) Open Solver Interface (OSI) (a uniform API for calling math programming solvers), and may be linked to any solver with an OSI interface (currently CLP, CPLEX, dylp, GLPK, OSL, SOPLEX, VOL

and XPRESS-MP). A FLOPC++ model has a pointer to an OSI object through which the methods of its class can be invoked (using the overloaded dereferencing operator `->`). This can be useful, for example, to output an MPS file of a problem or to modify it without regenerating the problem.

3 Conclusion

The syntax and expressiveness of FLOPC++ for linear optimization modelling is similar to traditional modelling languages, but its implementation as a C++ class library implies several additional benefits:

- The relatively small size of the source code makes changes and extensions easy.
- Fast problem generation. Sparse generation efficiently implemented and problem generation done by compiled code.
- Seamless integration with applications.
- Convenient and efficient implementation of model tailored solution algorithms. Model modification (without regeneration) available.
- Modelling objects are first class C++ types (i.e. they can be passed around as parameters to functions). This allows higher level modelling extensions (such as a "minimize max" objective function, etc.) to be implemented.