Math. Prog. Comp. (2009) 1:223–247
DOI 10.1007/s12532-009-0008-2

FULL LENGTH PAPER

A structure-conveying modelling language for mathematical and stochastic programming

Marco Colombo · Andreas Grothey · Jonathan Hogg · Kristian Woodsend · Jacek Gondzio

Received: 24 March 2009 / Accepted: 20 October 2009 / Published online: 11 November 2009

Abstract We present a structure-conveying algebraic modelling language for mathematical programming. The proposed language extends AMPL with object-oriented features that allows the user to construct models from sub-models, and is implemented as a combination of pre- and post-processing phases for AMPL. Unlike traditional modelling languages, the new approach does not scramble the block structure of the problem, and thus it enables the passing of this structure on to the solver. Interior point solvers that exploit block linear algebra and decomposition-based solvers can therefore directly take advantage of the problem's structure. The language contains features to conveniently model stochastic programming problems, although it is designed with a much broader application spectrum.

Mathematics Subject Classification (2000) 68N15 · 90C06 · 90C15

Supported by: "Conveying structure from modeling language to a solver", Intel Corporation, Santa Clara, USA.
M. Colombo was supported by EPSRC grant EP/E036910/1.

M. Colombo (✉) · A. Grothey · J. Hogg · K. Woodsend · J. Gondzio
School of Mathematics and Maxwell Institute, University of Edinburgh, Edinburgh, Scotland
e-mail: M.Colombo@ed.ac.uk

A. Grothey
e-mail: A.Grothey@ed.ac.uk

J. Hogg
e-mail: J.Hogg@ed.ac.uk

K. Woodsend
e-mail: K.Woodsend@ed.ac.uk

J. Gondzio
e-mail: J.Gondzio@ed.ac.uk

1 Introduction

Algebraic modelling languages (AMLs) are recognised as an important tool in the formulation of mathematical programming problems. They facilitate the construction of models through a language that resembles mathematical notation, and offer convenient features such as automatic differentiation and direct interfacing to solvers. Their use vastly reduces the need for tedious and error-prone coding work. Examples of popular modelling languages are AMPL [13], GAMS [5], AIMMS [3] and Xpress-Mosel [7]. See [14,23,31] for surveys.

The increased availability of computing power (through multiprocessor architectures) and the advances in the implementation of solvers (through structure exploitation) mean that users expect the solution of ever-growing problem instances to be viable. It has long been recognized that most large-scale problems are sparse and that exploiting sparsity is key to their numerical tractability. Modelling languages therefore pass information about the sparsity of a problem to a solver. However, today's large-scale optimization problems are typically not only sparse but also structured. By *structure* we mean the presence of a discernible pattern in the non-zero entries of the constraint matrix. In such cases, it is often possible to partition the matrix into blocks, only some of which contain non-zero elements, while the rest are empty. These structures usually appear in the problem (and the problem matrices) due to some underlying generating process. In particular, they may reflect a discretization process over time (as in optimal control problems), over space (for PDE-constrained optimization) or over a probability space (as in stochastic programming). Further, they might reflect a structure inherent to the concept being modelled (e.g. a company consisting of several divisions). Often these structures are also nested.

In most cases, the structure (or at least the process generating it) is known to the modeller. Many approaches to solving large-scale optimization problems, such as decomposition techniques and interior point methods, can efficiently exploit the structure present in a problem. Therefore, it seems natural to pass the knowledge about the structure from the modeller to the solver. However, current modelling languages do not, in general, offer this possibility. While some languages have features to express network constraints (and pass this information to the solver), and there are some specialised input formats (such as the SMPS format for stochastic programming), most of the currently available algebraic modelling languages do not offer the flexibility to model general nested structures, preserve the structure during the problem generation and feed it to the solver. A notable exception is provided by modelling languages based on the constraint programming paradigm [30,26].

Further, truly large scale problems may require parallel processing not only for the solution of the problem, but also during the model generation phase. The Asset and Liability Management problem solved in [17] requires 60 GB of memory just to store the matrices and vectors describing the problem; even the raw data used to generate the matrices requires 2 GB of storage. It would be impossible for a modelling tool to construct the solver files if it considered the problem as a whole. An AML that facilitates the modelling of the problem structure allows the development of a parallel model generator for the language which would overcome these problems. We explored this issue in [21], in which an earlier version of the structure-conveying language is described.

This paper addresses these issues by proposing an extension to AMPL that allows the modeller to express the structure inherent to the problem in a natural way. The AMPL language is extended by the `block` keyword that groups together model entities and allows them to be repeated over indexing sets. Compared with our preliminary development [21], in this paper we present the extensions for stochastic programming with recourse. An implementation of the interpreter for the language extensions accompanies this paper; in addition we provide an implementation of the AMPL solver interface for the OOPS solver [20]. Together they allow the full journey from structured model to solution. It should be noted that the design is generic, and allows the use of other structure-exploiting solvers.

The paper is organised as follows: In the next section we review some background on mathematical programming and existing approaches to modelling structured mathematical programming problems. Section 3 presents the design and syntax of our structure-conveying modelling language (SML), using the example of a survivable network design problem to illustrate the discussion. Section 4 shows how SML deals with the special case of stochastic programming, using an Asset and Liability Management example. Section 5 describes the design of the solver interface, and provides more information on our implementation, before we draw our conclusions in Sect. 6.

2 General and structured optimization problems

A mathematical programming problem can be written in the form

$$\min_x f(x) \quad \text{s.t. } g(x) \leq 0,\ x \geq 0,\ x \in \mathcal{X}, \tag{1}$$

where $\mathcal{X} \subset \mathbb{R}^n$, $f : \mathbb{R}^n \to \mathbb{R}$, $g : \mathbb{R}^n \to \mathbb{R}^m$. Functions f and g are usually assumed to be sufficiently smooth: $f,\ g \in C^2$ is common. The vector-valued constraint function g can be thought of as vector of scalar constraints $g(x) = (g_1(x), \ldots, g_m(x))^T$. The set $\mathcal{X}$ from which the decision variables are taken can include simple bounds on the variables or further restrictions such as integrality constraints. A solution algorithm for problems as general as (1) will typically require access to routines that evaluate $f(x), g_i(x), \nabla f(x), \nabla g_i(x), \nabla^2 f(x), \nabla^2 g_i(x)$ at various points $x \in \mathcal{X}$.

Mathematical programming can frequently reach very large sizes: n, m being of the order of millions is not uncommon. For problems of these sizes, the variables x and constraint functions $g_i(x)$ are usually generated by replicating a limited number of variable and constraint prototypes over large index sets. This repetition of prototypes over large sets, typical of large-scale mathematical programming problems, leads to matrices in which the sparsity pattern displays a structure. We are particularly interested in structures where the Hessian matrix $\nabla^2 f(x)$ has non-zero entries contained in diagonal blocks, while for the Jacobian matrix of the constraints $\nabla g(x)$ non-zero entries lie in some combination of diagonal, row and column blocks.

Several efficient solution approaches exist that are capable of exploiting such a block structure. Generally speaking, they can be classified into decomposition approaches and interior point methods. Both classes have the strength that they parallelize very well. One possible advantage of interior point methods is their broad applicability to

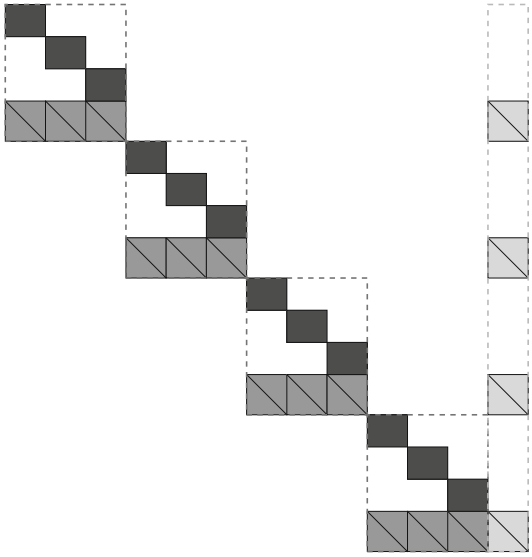


Fig. 1 Structure of the constraint matrix for the MSND problem

nonlinear as well as linear problem formulations, while decomposition-based techniques encounter stronger challenges in the nonlinear case.

2.1 Solution approaches to structured optimization problems

In a decomposition framework (e.g. Benders decomposition [1] or Dantzig-Wolfe decomposition [8]) the problem is decomposed into two (or more) levels. In the top level, the primal values for complicating variables (linking columns) or the dual values for complicating constraints (linking rows) are temporarily fixed; this results in a separation of the problem into many smaller subproblems (corresponding to the diagonal blocks in Fig. 1) that can be solved independently. The top level optimises these complicating primal/dual variables by building up value surfaces for the optimal subproblem solutions.

Within an Interior Point Method (IPM), the main work of the algorithm is to solve systems of equations with the augmented system matrix $\Phi = \begin{bmatrix} -(H + \Theta^{-1}) & A^T \\ A & 0 \end{bmatrix}$ where $H = \nabla^2 f(x) + \sum_{i=1}^{m} y_i \nabla^2 g_i(x)$ is the Hessian of the Lagrangian, Θ is a diagonal scaling matrix, and $A = \nabla g(x)$ is the Jacobian of the constraints. For problems with a block-structured A matrix, the linear algebra computations required by the algorithm can be organized to exploit the structure [4,28]. Alternatively, the matrix Φ itself can be reordered into a block-structured matrix [20], which can then be exploited in the linear algebra of the IPM using Schur complement techniques and the Sherman-Morrison-Woodbury formula. The object-oriented parallel interior point solver OOPS [19] does this by building a tree of submatrices, with the top node corresponding to the whole problem and the nodes further down the tree corresponding to the constituent parts of the problem matrices. All linear algebra operations needed for the interior

point algorithm (such as factorisations, backsolves, and matrix-vector products) can be broken down into operations on the tree: i.e. a factorisation on the top node of the tree can be broken down into factorisations, solves, and matrix-vector products involving nodes further down the matrix tree. This approach generalises easily to more complicated nested structures and facilitates an efficient parallelization of the IPM. The power of this approach has been demonstrated by solving an Asset and Liability Management problem with over 10^9 variables on 1,280 processors in under 1 h [17].

In the rest of the paper, we will forgo possible limitations of applicability of our modelling language to the general nonlinear problem (1), which would needlessly encumber us with assumptions, restrictions or details that are outside the scope of this paper, and will concentrate on the modelling of linear and quadratic problems. The design of the proposed modelling framework is kept general enough to embrace nonlinear optimization problems. However, to reflect the current status of the implementation and for ease of presentation of both mathematical formulations and corresponding model descriptions, we have restricted our examples to linear and quadratic programming problems.

2.2 Modelling approaches to structured problems

While it is possible to implement routines to evaluate the functions f, g_i of (1) and their derivatives in a programming language such as C or Fortran, this is a tedious and error-prone task. An Algebraic Modelling Language (AML) provides facilities to express a mathematical programming problem in a form that is both close to the mathematical formulation and easy to parse by an automated tool. The AML tool will then provide the solver with the required information about the functions that define the problem.

Sparsity has long been recognised as an important factor in solving optimization problems efficiently, hence all modelling languages pass sparsity information to a solver. To apply any structure-exploiting solution approach, however, additional information about the block structure of the problem must be provided to the solution algorithm. This information is not immediately apparent from usual AML models. Moreover, AMLs typically order rows and column of the generated system matrices by constraint and variable names rather than by structural blocks, thus losing information about the problem structure. While there have been attempts to automatically recover such a structure from the sparsity pattern of the problem matrices [9], these approaches are generally computationally expensive and they obtain far from perfect results. Such procedures ignore the essential fact that in most cases the structure of the problem (or at least the process generating the structure) is known to the modeller, but it is lost in the model generation process for the lack of appropriate language constructs. The proper remedy is to enable the modeller to express the structural description of the problem directly in the modelling language. The need for a structure-conveying modelling language has been raised several times by practitioners, especially those involved in stochastic programming [31]. However there are very few actual implementations and no universally accepted approach exists. In what follows, we briefly review the most relevant attempts.

An early effort was the SMPS format for stochastic programming problems [2]. SMPS is an extension of the standard MPS format [25] (created to represent LP problems) that is targeted at the special structures encountered in stochastic programming. Being a specialized data format rather than a modelling language, it produces problem instances that are cumbersome to write, hard to read, and laborious to modify.

Many different modelling tools specifically targeted at stochastic programming have been suggested, and we mention the most relevant. sMAGIC [6] uses a recursive model formulation that can be exploited in nested Benders decomposition, but unfortunately it is limited to linear programming problems. SAMPL [29] and the proposed stochastic programming extension to AMPL [10] add new language constructs to the ones existing in the current version of AMPL with the aim to ease the formulation of several classes of stochastic programming problems. StAMPL [12] and MusMod [22] separate the model at a time stage from the stochastic tree topology, that is described as data. Our approach is similar in spirit to theirs.

There have been very few attempts at producing a general structure-conveying modelling language. SET [15] offers facilities to declare the structure of a model as a postscript to the problem: an additional structure file defines the blocks making up the structure of the problem and a list of row and column names that make up each block. This approach requires that the full (unstructured) problem be processed first by the modelling language: hence this hampers any potential parallelisation of the model generation. Furthermore, considerable (and in our opinion needless) effort goes into unscrambling the problem following the structure definition file. The authors of AMPL argue in [11] that AMPL's suffix facility can be used to the same effect (but with the same restrictions). Finally, MILANO [27] is an interesting concept of an object-oriented modelling language; however its available documentation is scarce and it seems to have been abandoned.

Overall, the main concern of the described approaches is with easing the work for the modeller by supplementing the syntactic capabilities of existing AMLs, rather than conveying additional structural information to the solver or allowing the parallelization of the generation process itself.

3 Design of a structure-conveying modelling language

AMPL [13] is a widely used algebraic modelling language. One of its main advantages is its well-defined interface that makes the linking to new solver backends straightforward. This linking is done through the `amplsolver` library [16], that provides routines to read AMPL's binary nl-file model representation. For these reasons we have based our structure-conveying modelling language (SML) on AMPL, and extended AMPL's capabilities by providing keywords to express the problem structure. While the current implementation is based on AMPL, which we follow quite closely in terms of language extension, the ideas presented in this paper (mainly, the `block` keyword) are general and apply to possible extensions based on different algebraic modelling languages.

We will use a multi-commodity survivable network design (MSND) problem as the motivating example throughout this section. Through this example we aim to show

how existing languages are not suited to capturing the problem structure in a natural way.

3.1 Example problem: survivable network design

A network is described by sets $\mathcal{V}$ of vertices and $\mathcal{E}$ of (directed) edges and a base capacity C_j for every edge $j \in \mathcal{E}$. The basic multi-commodity network flow (MCNF) problem considers the best way to move commodities $k \in \mathcal{C}$ from their sources to their respective destinations, through the edges of the shared-capacity network. Each commodity is described as the triplet (s_k, t_k, d_k) consisting of start vertex s_k, terminal vertex t_k and amount to be shipped d_k. A feasible flow $x_k = (x_{k,j})_{j\in\mathcal{E}}$ for the kth commodity can be represented by the constraint

$$Ax_k = b_k, \quad x_k \geq 0,$$

where $A \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{E}|}$ is the vertex–edge incidence matrix, and $b_k = (b_{k,i})_{i\in\mathcal{V}}$ is the demand vector for the kth commodity, with the following conventions:

$$A_{i,j} = \begin{cases} -1 & \text{vertex } i \text{ is source of edge } j, \\ 1 & \text{vertex } i \text{ is target of edge } j, \\ 0 & \text{otherwise,} \end{cases} \quad b_{k,i} = \begin{cases} -d_k & \text{vertex } i \text{ is source of demand } k, \\ d_k & \text{vertex } i \text{ is target of demand } k, \\ 0 & \text{otherwise.} \end{cases}$$

In multi-commodity survivable network design (MSND) the aim is to find the minimum installation cost of additional (spare) capacities S_j at price c_j, $j \in \mathcal{E}$, so that the given commodities can still be routed through the network if any one edge or vertex should fail. The MSND problem can be modelled by a series of multi-commodity network flow problems, in each of which one of the original edges or vertices is removed. Note that the subproblems are not completely independent, as they are linked by the common spare capacities S_j.

Let $A^{(e,l)}$, $l \in \mathcal{E}$, be the vertex–edge incidence matrix in which the column corresponding to the lth edge is set to zero. Then any vector of flows $x_k^{(e,l)} \geq 0$ satisfying

$$A^{(e,l)}x_k^{(e,l)} = b_k, \quad k \in \mathcal{C},\ l \in \mathcal{E},$$

gives a feasible flow to route the kth commodity through the edge-reduced network. Similarly let $A^{(v,i)}$, $i \in \mathcal{V}$, be the vertex–edge incidence matrix in which the row corresponding to the ith vertex and the columns corresponding to edges incident to this vertex are set to zero. Then any vector of flows $x_k^{(v,i)} \geq 0$ satisfying

$$A^{(v,i)}x_k^{(v,i)} = b_k, \quad k \in \mathcal{C},\ i \in \mathcal{V},$$

gives a feasible flow to route the kth commodity through the vertex-reduced network. As the network is capacity-limited, however, each edge $j \in \mathcal{E}$ can carry at most $C_j + S_j$

units of flow. The complete formulation of the MSND problem is as follows:

$$
\begin{aligned}
\min \quad & \sum_{j \in \mathcal{E}} c_j S_j \\
\text{s.t.} \quad & A^{(e,l)} x_k^{(e,l)} = b_k && \forall k \in \mathcal{C}, l \in \mathcal{E} \\
& A^{(v,i)} x_k^{(v,i)} = b_k && \forall k \in \mathcal{C}, i \in \mathcal{V} \\
& \sum_{k \in \mathcal{C}} x_{k,j}^{(e,l)} \leq C_j + S_j && \forall j \in \mathcal{E}, l \in \mathcal{E} \\
& \sum_{k \in \mathcal{C}} x_{k,j}^{(v,i)} \leq C_j + S_j && \forall j \in \mathcal{E}, i \in \mathcal{V} \\
& x \geq 0, \;\; S_j \geq 0 \quad \forall j \in \mathcal{E}
\end{aligned}
\tag{2}
$$

The constraint matrix of this problem has the form shown in Fig. 1. The basic building blocks are the vertex–edge incidence matrices (shown in dark grey). These blocks are repeated for every commodity that needs to be shipped, and they are linked by a series of common rows (shown in medium grey) that represent the global capacity constraints to build a MCNF problem. Each MCNF block (framed by dashed lines) is repeated for every (missing) edge and vertex. The common capacity variables (light grey blocks) act as linking columns. While the nested structure of the problem is obvious is Fig. 1, it cannot be easily appreciated from the mathematical formulation (2).

3.2 Standard modelling language formulation

Problem (2) can be represented in an algebraic modelling language as shown in Fig. 2, where we adopt a syntax that is very close to AMPL, slightly modified to compress the example and aid readability; models in other languages will look much the same.

The modelling language approach has the desirable property of separating model and data, but this formulation has a few notable deficiencies.

First of all, the model appears overly complicated and hardly elegant: the flow variables have to be indexed over three sets to be able to capture the commodity they refer to, the edges they traverse, and which edge or vertex is currently missing from the network; the flow balance constraints are also indexed over three sets and require a nested indexing over set `EDGES`, which makes the intentions unclear; the conditions on some of the sets that describe the modified network (such as `{j in EDGES:j~=e, edge_target[j]==i}`) need to be restated several times, are difficult to read and make the coding phase error-prone. Other approaches to modelling this problem (such as defining edges as pairs of vertices) may improve readability.

A further, less obvious, shortcoming should be noted: some of the `Flow[l,k,j]` variables (those for which `l==j`, corresponding to the flow over the missing edge), are not strictly required, even though they appear in constraints. This is not made explicit in the model, and the model formulation as written will result in empty columns. This could of course be remedied: either by a revised model, at the expense of significantly

```
set VERTS, EDGES, COMM;
param cost{EDGES}, basecap{EDGES};
param edge_source{EGDES}, edge_target{EDGES};
param comm_source{COMM}, comm_target{COMM}, comm_demand{COMM};
param b{k in COMM, i in VERTS} :=
  if  (comm_source[k]==i) then comm_demand[k] else
    if (comm_target[k]==i) then -comm_demand[k] else 0;

# first index is missing egde/vertex, then commodity, then edge of flow
var Flow{EDGES union VERTS, COMM, EDGES} >= 0;
var sparecap{EDGES} >= 0;

# flow into vertex - flow out of vertex equals demand
subject to FlowBalanceMissingEdges{e in EDGES, k in COMM, i in VERTS}:
  sum{j in EDGES:j~=e, edge_target[j]==i} Flow[e,k,j]
   - sum{j in EDGES:j~=e, edge_source[j]==i} Flow[e,k,j] = b[k,i];

subject to FlowBalanceMissingVerts{v in VERTS, k in COMM, i in VERTS diff {v}}:
  sum{j in EDGES:edge_target[j]==i, edge_source[j]~=v} Flow[v,k,j]
   - sum{j in EDGES:edge_source[j]==i, edge_target[j]~=v} Flow[v,k,j] = b[k,i];

subject to Capacity{ev in EDGES union VERTS, j in EDGES}:
   sum{k in COMM} Flow[ev,k,j] <= basecap[j] + sparecap[j];

minimize obj: sum{j in EDGES} sparecap[j]*cost[j];
```

Fig. 2 AMPL-like model for the MSND problem

complicating the formulation, or by an automated process (in the model translator, as done by AMPL, or as a pre-processing phase of the solver) spotting and removing them. Nevertheless, we consider this lack of expressive power to be a deficiency in the modelling approach.

Finally, as with the mathematical description (2), the structure of the problem is not exposed from the modelling language formulation. As such, the model processor is incapable of generating a constraint matrix in a structured fashion.

3.3 SML formulation

The main contribution of SML is the introduction of the `block` keyword, that is used to define a sub-model. Its power comes from the fact that a `block` is indexable, and thus it can be conveniently repeated. A sub-model definition using the `block` command takes the form:

```
block nameofblock{j in nameofset} : {
  ...
}
```

Within the scope delimited by `block { ... }`, any number of `set, param, subject to, var, minimize` or indeed further `block` definitions can be

placed. The interpretation is that all declarations placed inside the block environment are implicitly repeated over the indexing expression used for the `block` command. Clearly, the nesting of such `blocks` creates a tree structure of blocks.

A `block` command creates a variable *scope*: within it, entities (sets, variables, constraints) defined inside this block or in any of its parents can be used. Entities defined in the block can be accessed from outside by using the form `nameofblock[j].nameofentity` inspired by the naming convention of object-oriented programming. Entities defined in nodes that are not direct children or ancestors of the current block cannot be referenced as this would make the resulting structure more difficult to exploit.

Using the `block` keyword, the MSND model of Fig. 2 can be rewritten as shown in Fig. 3. There are several advantages of the SML formulation over the plain AMPL formulation. Firstly, the confusing triple indexing of the `Flow` variables and `FlowBalance` constraints is not needed any more, making the model much clearer to read. The fact that some of the `Flow` variables and some of the `FlowBalance` constraints are indeed empty is evident through the use of the `EDGEDIFF` and `VERTDIFF` sets. Most importantly, the nested structure of the problem (Fig. 1) is immediately apparent from the SML model file. Apart from making the structure exploitable by the solver, this also aids the modeller to visualize the relation between different model components.

It is worth mentioning how objectives are handled in SML. The model in Fig. 3 has a single objective declaration in the top level model. However often it is more convenient to place objectives (or partial objectives) in a submodel block. SML allows the placement of an objective declarations (through a `minimize` or `maximize` statement) anywhere in the model. As in plain AMPL, objectives are named. Objective declarations (in different submodels) that define an objective of the same name are summed together to define a global objective declaration of the given name. If more than one global objective function is defined, then by default the one given last in the top level model is used. The objective direction (minimize/maximize) is determined by the declaration given at the top level of the model (if there is none, the default is minimize). Any declaration with a differing direction to that at the top level is added to the global objective after being multiplied by -1.

3.4 The "prototype" and the "expanded" model trees

The block definitions in the SML model (Fig. 3) can be described by a tree (see left-hand side in Fig. 4). This is what we call the *prototype model tree*. It has one node for each type of block rather than for each actual block present in the problem. The prototype tree is obtained from the SML model file, and gives a compact representation of the problem.

Also the dependency between the blocks that make up the structure of the MSND problem of Fig. 1 can be described by a tree (see right-hand side of Fig. 4, for an example with two nodes, two arcs and three commodities), that we call the *expanded model tree*. The root node of the tree is the whole problem, and its children are the

```
set VERTS, EDGES, COMM;
param cost{EDGES}, basecap{EDGES};
param edge_source{EDGES}, edge_target{EDGES};
param comm_source{COMM}, comm_target{COMM}, comm_demand{COMM};
param b{k in COMM, i in VERTS} :=
  if  (comm_source[k]==i) then comm_demand[k] else
    if (comm_target[k]==i) then -comm_demand[k] else 0;

var sparecap{EDGES} >= 0;

block MCNFEdge{e in EDGES}: {
  set EDGEDIFF = EDGES diff {e};  # local EDGES still present

  block Net{k in COMM}: {
    var Flow{EDGEDIFF} >= 0;
    # flow into vertex - flow out of vertex equals demand
    subject to FlowBalance{i in VERTS}:
      sum{j in EDGEDIFF:edge_target[j]==i} Flow[j]
      - sum{j in EDGEDIFF:edge_source[j]==i} Flow[j] = b[k,i];
  }

  subject to Capacity{j in EDGEDIFF}:
     sum{k in COMM} Net[k].Flow[j] <= basecap[j] + sparecap[j];
}

block MCNFVert{v in VERTS}: {
  set VERTDIFF = VERTS diff {v}; # local VERTS still present
  set EDGEDIFF = {i in EDGES: edge_source[i]~=v, edge_target[i]~=v};

  block Net{k in COMM}: {
    var Flow{EDGEDIFF} >= 0;
    # flow into vertex - flow out of vertex equals demand
    subject to FlowBalance{i in VERTDIFF}:
      sum{j in EDGEDIFF:edge_target[j]==i} Flow[j]
      - sum{j in EDGEDIFF:edge_source[j]==i} Flow[j] = b[k,i];
  }

  subject to Capacity{j in EDGEDIFF}:
     sum{k in COMM} Net[k].Flow[j] <= basecap[j] + sparecap[j];
}

minimize costToInstall: sum{j in EDGES} sparecap[j]*cost[j];
```

Fig. 3 SML model for the MSND problem

MCNF problems for all possible choices of the missing arcs and nodes. At the bottom level, the leaf nodes in the tree are the network routing problems: each MCNF node contains one of these for each commodity.

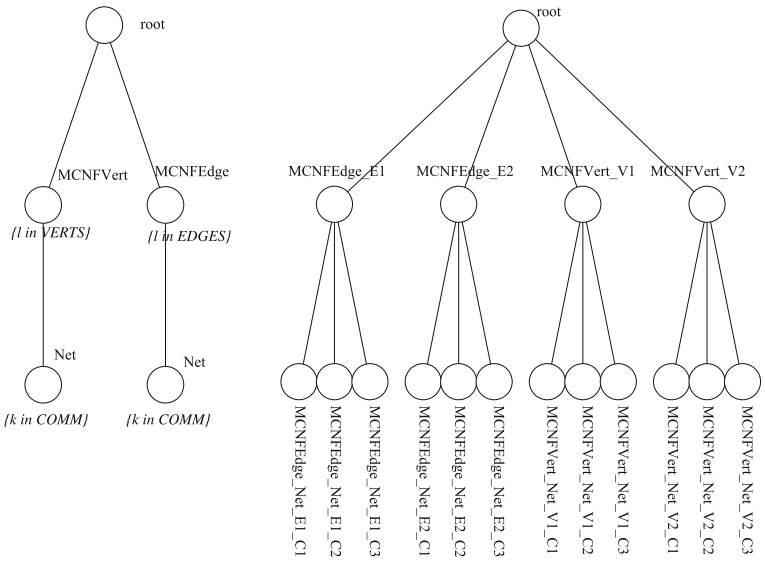


Fig. 4 Prototype and expanded model trees for the MSND problem

The expanded tree (and therefore the full problem) can be obtained from the prototype tree by repeating nodes according to the indexing expression used in the definition of the block.

4 Stochastic programming in SML

Uncertainty in the data is a commonly observed phenomenon in real-life optimization problems. It can be argued that nearly all practical optimization problems display uncertainty in the data, even if this is not made explicit in the chosen solution method. Stochastic programming is a popular modelling approach for problems involving uncertainty.

Stochastic programming describes situations in which information about the problem parameters becomes available at various points in time. The decision makers have to decide on some course of action (first stage decisions) before the full information is known; as the information becomes available, they can take different actions (second stage decisions/recourse actions) that are allowed to depend on the observed event. In a multistage setting there may be several cycles of observing events and taking recourse actions. For a correct representation of the time structure, it is essential that the model guarantees that decisions up to a point in time can depend only on the information available up to that point, and not on any information to be revealed at later stages (nonanticipativity). Figure 5 (taken from [12]) shows a typical decision process.

While the underlying stochastic process that generates the events may be continuous, it is customary, in order to obtain a computationally viable description, to work with a discretized approximation. Such a discretized decision process is commonly represented by a scenario tree (Fig. 6).

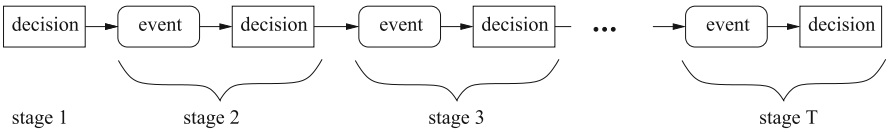
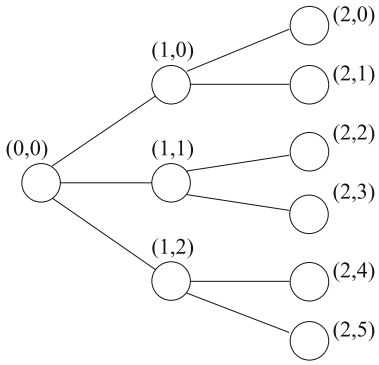


Fig. 5 Stochastic programming decision process

Fig. 6 A scenario tree



Each level in the tree corresponds to a point in time when some of the information is revealed and a consequent action can be taken. Each node corresponds to a series of event realisations that have been observed up to this point in time. The branches that depart from a node correspond to possible future realisations that we are considering from that particular node. To each node of the tree we can assign a probability of being reached. The size of the resulting problem is exponential in the number of stages, leading quickly to very large problem dimensions.

Stochastic programming is an important source of highly structured problems, particularly in the multistage setting. The structure is very well understood, and modellers have a good deal of control over it, and again exploiting this structure is the key to efficient solution approaches. Therefore, it is fundamental that the way a stochastic programming problem is formulated can convey this structure to the solver.

4.1 Example problem: asset and liability management

As an example of a stochastic programming problem we use an asset and liability management problem [18]. The problem is concerned with investing an initial cash amount b into a given set of assets $\mathcal{A}$ over T time periods in such a way that, at every time stage $0 < t \leq T$, a liability payment l_t can be covered. Each asset has value v_j, $j \in \mathcal{A}$. The composition of the portfolio can be changed throughout the investment horizon, incurring (proportional) transaction costs c_t, $x^h_{i,j}$ is the amount of asset j held at node i, $x^b_{i,j}$ and $x^s_{i,j}$ are the amounts bought and sold. These satisfy the inventory and cash balance equations at every node. The random parameters are the asset returns $r_{i,j}$ that hold for asset j at node i. The evolution of uncertainties can be described

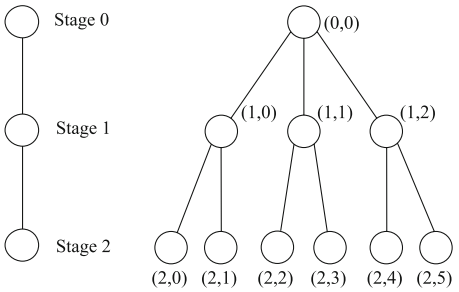


Fig. 7 Prototype tree and expanded tree for stochastic programming

by an event tree: For a node i, $\pi(i)$ is its parent in the tree, p_i is the probability of reaching it, and $\tau(i)$ represents its stage. With $\mathcal{L}_T$ we denote the set of final-stage nodes. The root node of the tree is assumed to have index $t = 0$. The objective maximises a linear combination of the expectation of the final portfolio value (denoted by μ) and its negative variance, with risk-aversion parameter λ, as in the Markovitz model [24]:

$$\max\{\mathbb{E}(\text{wealth}) - \lambda \text{Var}(\text{wealth})\} = \max\left\{\mathbb{E}\left(\text{wealth} - \lambda\left[\text{wealth}^2 - \mathbb{E}(\text{wealth})^2\right]\right)\right\}.$$

The complete model is the following:

$$\begin{aligned}
\max_{x,\,\mu\geq 0}\ & \mu - \rho\left[\sum_{i\in\mathcal{L}_T} p_i\left[(1-c_t)\sum_{j\in\mathcal{A}} v_j x^h_{i,j}\right]^2 - \mu^2\right] \\
\text{s.t.}\ & x^h_{i,j} = (1+r_{i,j})x^h_{\pi(i),j} + x^b_{i,j} - x^s_{i,j}, && \forall i \neq 0,\ j\in\mathcal{A} \qquad (3)\\
& \sum_{j\in\mathcal{A}} (1-c_t)v_j x^s_{i,j} = l_{\tau(i)} + \sum_{j\in\mathcal{A}} (1+c_t)v_j x^b_{i,j}, && \forall i \neq 0 \\
& \sum_{j\in\mathcal{A}} (1+c)v_j x^b_{0,j} = b \\
& \sum_{i\in\mathcal{L}_T} p_i \sum_{j\in\mathcal{A}} v_j x^h_{i,j} = \mu
\end{aligned}$$

4.2 SML formulation

The model structure displayed by stochastic programming problems such as (3) is naturally suited to being expressed with SML. The different decision stages of Fig. 5 can be represented as a nested chain of submodels. This corresponds, in our terminology, to the prototype tree of the model (Fig. 7, left). Each submodel, i.e. each node of the prototype tree, is indexed by the set of random events at this stage, thus producing an expanded tree (Fig. 7, right). Note the exact correspondence between expanded tree and the scenario tree of Fig. 6.

```
param Budget, Ct, Rho, Liability1, Liability2;
set ASSETS, NODES;
param Parent{NODES}
param Value{ASSETS};
param Ret{NODES, ASSETS};
param Probs{NODES};
var mu;

# Stage 0:
var xh0{ASSETS} >= 0, xb0{ASSETS} >= 0;
subject to StartBudget:
  (1+Ct)*sum{j in ASSETS} xb0[j]*Value[j] <= Budget;

# Stage 1:
block Stage1{n1 in NODES:Parent[k]==0}: {
  var xh1{ASSETS} >= 0, xb1{ASSETS} >= 0, xs1{ASSETS} >= 0;
  subject to Inventory{j in ASSETS}:
    xh1[j] = (1+Ret[n1,j]) * xh0(j) + xb1[j] - xs1[j];
  subject to CashBalance:
    (1-Ct) * sum{j in ASSETS} Value[j]*xs1[j] =
      Liability1 + (1+Ct) * sum{j in ASSETS} Value[j]*xb1[j];

  # Stage 2:
  block Stage2{n2 in NODES:Parent[n2]==n1}: {
    var xh2{ASSETS} >= 0, xb2{ASSETS} >= 0, xs2{ASSETS} >= 0;
    subject to Inventory{j in ASSETS}:
      xh2[j] = (1+Ret2[n2,j]) * xh1(j) + xb2[j] - xs2[j];
    subject to CashBalance:
      (1-Ct) * sum{j in ASSETS} value[j]*xs2[j] =
        Liability2 + (1+Ct) * sum{j in ASSETS} Value[j]*xb2[j];

    var wealth := Prob[n1] * Prob[n2] * sum{j in ASSETS} Value[j]*xh2[j];

    maximize objFunc:
      ( mu - Rho * ((wealth*wealth) - mu*mu) ) * Prob[n1] * Prob2[n2]
  }
}

subject to ExpPortfolioValue:
  mu = sum{n1 in NODES:Parent[n1]==0, n2 in NODES:Parent[n2]==n1}
     Prob[n1] * Prob[n2] * Stage1[n1].Stage2[n2].wealth;
```

Fig. 8 Explicitly-blocked model for a simple 3-stage asset liability problem

The stochastic programming problem (3) could be modelled in SML using the `block` keyword (and a three-stage model written explicitly this way is shown in Fig. 8). However, such an approach ignores two important considerations:

1. The problem description for a stochastic programming problem is typically very similar for each stage (with possibly more accentuated differences for the first and last stage): a modelling approach where each stage is written out explicitly would necessarily present redundant information, increasing the chance of error and the size of the model's SML representation.
2. The shape of the scenario tree (including the number of stages) should be treated as problem data: instead, writing each stage explicitly as blocks forces the scenario tree to be hard-coded into the model.

Our approach to the modelling of stochasticity in SML is guided by the observation that the stage-wise model description and the specification of the stochasticity are two logically separate phases that should not be coupled. The modeller is first interested in defining the multistage setting and in laying out how the stages interact with each other. The modelling of stochasticity is done independently at a later time; moreover, there can be many different event trees associated to the same multistage model, thus reinforcing the need to separate this aspect from the model proper.

SML provides language facilities to describe the stages within a stochastic program and it allows the definition of how the model changes at different stages. Information on how the model is to be represented as prototype and expanded model tree (i.e. the number of stages and the scenario tree) is supplied as data. As desired, this design makes it possible to have different event trees defined for the same model by only changing the problem data.

A stochastic program is declared in SML via the `stochastic` modifier to the `block` command:

```
block nameofblock stochastic using(NODESET, Prob, Parent,
  STAGESET) : {
  ...
}
```

The parameters provided in the (mandatory) `using` block describe the shape of the scenario tree. They are assumed to be declared as

```
set NODESET;
param Prob{NODESET};
param Parent{NODESET} symbolic;
set STAGESET ordered;
```

`STAGESET` and `NODESET` are the set of stages and nodes, respectively, `Prob` gives the conditional probability of reaching a node given that its parent was reached, and finally, `Parent` describes the tree structure by defining the parent for every node (set to the keyword `"null"` for the root node). It is assumed that the number of levels thus defined in the tree matches the cardinality of the `STAGESET` set. In keeping with the usual syntax, the declaration of dummy variables over the node and stage sets is permitted.

By default, all entities declared within the scope of `block stochastic` are instantiated for all nodes. However, declarations of model components can be limited to certain stages, either by a `stages` attribute:

```
var x{j in ASSETS} stages STAGESUBSET;
```

or as a separate block within the `block stochastic`:

```
stages STAGESUBSET: {
   ...
}
```

where `STAGESUBSET` is a subset of `STAGESET` that conveys information on how to index and replicate the entities declared within the `stages` block. Inside a `stages` block, we can nest one or more `blocks` if the problem has a further structure. Relations between different stages can be expressed by using the `ancestor(int i)` function that allows to reference the ith ancestor stage. Variables of the ancestor can be referenced using the normal SML syntax, such as `ancestor(i).x`.

Within a stochastic programming problem model, the language should allow the differentiation of time-dependent stochastic entities (the values of which depend on the realisation of the stochastic process, i.e. they are node dependent) from time-dependent deterministic entities (the values of which are known in advance). By default, an entity declared inside a `block stochastic` is considered to be completely stochastic, and its value is assumed to depend on the node of the event tree: therefore, there is one copy of such entity for each node. To identify a time-dependent deterministic entity, it can be declared as *deterministic* using either

`var` *varname* `deterministic;` or `deterministic:{...}`

resulting in only one copy of the entity existing at each time-stage (rather than at every node).

Objectives are treated in the same manner as before: elements of the objective in different blocks but having the same declared name are summed into one global objective. However, there is one important difference: in stochastic programming, the objective is always to minimize or maximize the expectation of an expression. Therefore, within a stochastic block, objective terms are always weighted by the unconditional probability of the node to which they belong. Conveniently, there is no need to specify the expectation explicitly, as the SML preprocessor performs this automatically.

As many stochastic programming models, such as the modelling of conditional value at risk (CVaR) or stochastic dominance constraints include expectation-like constraints, the language provides the `Exp(·)` function to compute the expected value of the parameter. The function operates across the nodes that are present in the stage where the function is called. `Exp(x)` is expanded internally into

`sum{nd in NODESET : nd in` *currentstage*`} Prob[nd] * x[nd]`

where *currentstage* is a list of nodes in the scope of the current `stages` block.

```
param Budget, T, Ct, Rho;
set ASSETS, NODESET, STAGESES := 0..T;
param Parent{NODESET} symbolic, Probs{NODESET};
param Value{ASSETS};
var mu;

block alm stochastic using (NODESET, Parent, Probs, STAGESET):{

  var xh{ASSETS} >= 0, xb{ASSETS} >= 0, xs{ASSETS} >= 0;
  param Ret{ASSETS};
  param Liability deterministic;

  stages {0}: {
    subject to StartBudget:
      (1+Ct)*sum{j in ASSETS} xb[j]*Value[j] <= Budget;
  }

  stages {1..T}: {
    subject to Inventory{j in ASSETS}:
      xh[j] = (1+Ret[j]) * ancestor(1).xh(j) + xb[j] - xs[j];
    subject to CashBalance:
      (1-Ct) * sum{j in ASSETS} Value[j]*xs[j] =
         Liability + (1+Ct)*sum{j in ASSETS} Value[j]*xb[j];
  }

  stages {T}: {
    var wealth := sum{j in ASSETS} Value[j]*xh[j];
    subject to ExpPortfolioValue:
      Exp(wealth) = mu;
    maximize objFunc: mu - Rho * ((wealth*wealth) - mu*mu )
  }
}
```

Fig. 9 Structured model using stochastic language features for the ALM problem

Figure 9 shows how the ALM model (3) can be modelled in SML, using the features of the language. The stochastic block definition can be seen as a concise description of the model chain respresentation of Fig. 8.

5 Software design and implementation

Although SML was initially conceived as a front-end to the object-oriented IPM solver OOPS, it is designed with as broad an application spectrum in mind as possible. In particular, it should be possible to use SML to communicate structured problems to

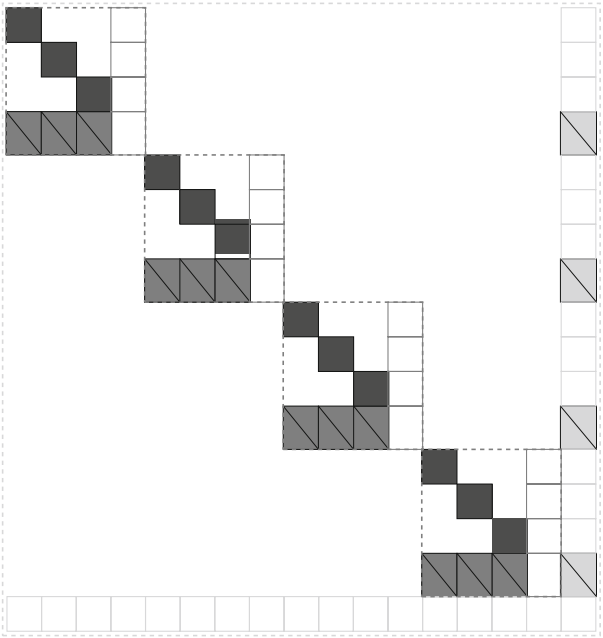


Fig. 10 System matrix for MSND problem as described by the expanded model tree: The root block (*light grey*) has only local variables (`sparecap`) but no local constraints (represented by *empty boxes*) and four children. Each of the children (*medium grey*, corresponding to the `MCNFEdge/Vert` blocks) has local constraints (`Capacity`) but no local variables and three children. Finally the bottom level blocks (*dark grey*, corresponding to the `Net` blocks) have both local variables and constraints but no children

any structure-exploiting solver, whether it be based on decomposition approaches or interior point methods (or something else). In this section we briefly review what model information is available from our description and then show how this fits naturally with the two most popular solution approaches (namely IPM and decomposition) and then proceed to describe how SML communicates this information to the solver. Our description refers to the MSND problem in Fig. 1 as an example.

The main vehicle for SML to describe a problem to a solver is the expanded model tree (Fig. 4). Each node of the expanded model tree represents a section of the model given by a set of local variables, local constraints and (if present) a list of child nodes. Figure 10 shows how the Jacobian of the MSND problem fits into this framework: note the use of empty boxes to denote blocks (local variables/constraints) that could be present at a particular node, but are not used in this particular model.

This representation of the structured model is natural for a solver using a nested linear algebra such as OOPS, that requires this information in two steps. In the first step, only a description of the shape of the expanded model tree is needed together with information on the dimension (number of rows and columns) in each of the leaf nodes of the tree; in the second stage, OOPS requires a separate description of each of the blocks in Fig. 10 (whether empty or not) as a sparse matrix. The information needed at the first stage is conveniently provided by SML through the expanded model tree. For the second stage, OOPS effectively needs to evaluate the Jacobian of

the local constraints of one expanded model node with respect to the variables local to another node. We will see that the same information is needed for a decomposition solver.

A decomposition solver applied to the MSND problem conceptually needs to solve the following nested problems:

$$\min_{S_j, j\in\mathcal{E}} \sum_{j\in\mathcal{E}} c_j S_j + \sum_{l\in\mathcal{E}} V_1^{(e,l)}(s) + \sum_{i\in\mathcal{V}} V_1^{(v,i)}(s) \tag{4}$$

$$\begin{aligned} V_1^{(e,l)}(s) &= \max_{\lambda} -\sum_j \lambda_j (C_j + S_j) + \sum_k V_2^{(k,e,l)}(\lambda), \quad l \in \mathcal{E} \\ V_1^{(v,i)}(s) &= \max_{\lambda} -\sum_j \lambda_j (C_j + S_j) + \sum_k V_2^{(k,v,i)}(\lambda), \quad i \in \mathcal{V} \end{aligned} \tag{5}$$

$$\begin{aligned} V_2^{(k,e,l)}(\lambda) &= \min_x \lambda^T x \quad \text{s.t. } A^{(e,l)} x = b_k \quad l \in \mathcal{E}, k \in \mathcal{C} \\ V_2^{(k,v,i)}(\lambda) &= \min_x \lambda^T x \quad \text{s.t. } A^{(v,i)} x = b_k \quad i \in \mathcal{V}, k \in \mathcal{C}. \end{aligned} \tag{6}$$

We leave aside considerations of whether problems (4)–(6) are infeasible and/or unbounded. Appropriate conditions and modifications to the problems to guarantee the well-definedness of the conceptual decomposition algorithm can be stated, but would distract from the purpose of this discussion.

Problem (6) needs to evaluate its system matrix $A^{(e,l)}/A^{(v,i)}$ (corresponding to the dark grey blocks in Fig. 10). Further, it needs to know how the Lagrange multiplier λ impacts on the current objective function, information obtained by evaluating the appropriate medium grey block. Similar reasoning for the other two problems (5) and (4) shows that a decomposition solver needs the capability to evaluate each of the coloured matrix blocks in Fig. 10 separately.

Following from this need to acquire the Jacobian in a blockwise manner, we provide separate functions to obtain each of the required block submatrices. For any pair of models between which an ancestor-descendant relation in the expanded model tree exists there is the potential for non-zero Jacobian block. In terms of the simplified schematic representation of the Jacobian structure in Fig. 11 a call to evaluate the intersection of a expanded tree node with itself will provide the block labelled A, a call to evaluate the intersection of an expanded tree node with any of its children will provide blocks labelled C and D (depending on the order of parent and child), whereas blocks B are obtained by intersecting a child node with itself.

We separate functions for returning the size of a block from those returning the entries of the block for two reasons:

Memory allocation To allow efficient allocation of available memory based on the sizes of the blocks.

Parallel planning To allow the solver to analyse the full model tree and distributing it to the correct processor before forcing generation of the data.

A key design decision was made to use AMPL itself to interpret the models. Therefore, SML is implemented in C++ as a pre- and post-processor to AMPL. The SML

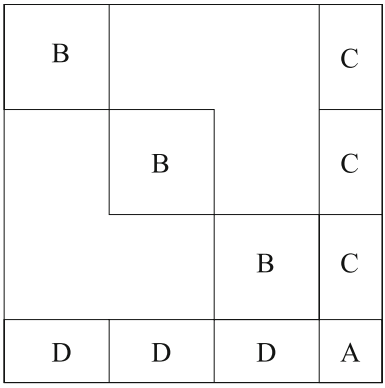


Fig. 11 Schematic representation of model blocks

model file is parsed to extract the prototype model-tree, the list of entities associated with each prototype-tree node and the dependency graph of the entities. The goal of the pre-processing stage is to create, for each node in the prototype model-tree, a stand-alone AMPL model file that describes the local model for this block of the problem. The file includes definitions for all the entities belonging to the prototype-tree node and all their dependencies: references to entities are changed to a global naming and indexing scheme, that is easily generated from the SML model file by generic text substitutions. Figure 12 shows these AMPL submodel files for the MSND problem formulation.

The AMPL submodels are then used to generate the appropriate submodels for every node of the expanded tree by changing the indexing sets. Each block definition is replaced by declaring an indexing set (named `*_SUB`) for the indexing expression of the sub-block, and this is prepended to the indexing expressions of every entity declared in the sub-block. By temporarily defining the set `EDGES_SUB` to a suitable subset of `EDGES` (leveraging AMPL's scripting commands), the model for any node on the expanded tree can be generated from the corresponding submodel `*.mod` file.

This process of model expansion based on indexing sets results in a `*.nl` file for every node of the expanded model tree; each file carries all the information about this node needed by a solver. The underlying submodel is the same for all expanded-tree nodes that correspond to the same prototype-tree node. However they are produced with different data instances: namely different choices of elements from the `block`'s indexing sets.

The nodes of the prototype and the expanded tree are internally represented as C++ objects that carry pointers to their children. Therefore, the prototype and expanded trees are themselves trees of C++ objects. The `ExpandedTreeNode` class provides information on the dimension of the node (number of local constraints and variables) and a list of its children; further, it provides methods to evaluate the Jacobian of its local constraints with respect to the local variables of a second `ExpandedTreeNode` object. Information on the number of sparse elements of this Jacobian block can be obtained prior to requesting the complete sparse matrix description to enable the allocation of sufficient memory. As argued above, these methods should satisfy

```
#------------------------------ root.mod --------------------------------
set EDGES;
param cost{EDGES};

var sparecap{EDGES} >= 0;

minimize obj: sum{j in EGDES} sparecap[j]*cost[j];
```

```
# ------------------------- root_MCNFEdge.mod ---------------------------
set EDGES, COMM;
param basecap{EDGES};

var sparecap{EDGES} >= 0;

set EDGES_SUB within EDGES;
set EDGEDIFF{e in EDGES_SUB} = EDGES diff {e};  # local EDGES still present

var MCNFEdge_Net_Flow{e in EDGES_SUB, EDGEDIFF[e], k in COMM} >= 0;
subject to Capacity{e in EDGES_SUB, j in EDGEDIFF[e]}:
     sum{k in COMM} MCNFEdge_Net_Flow[e,k,j] <= basecap[j] + sparecap[j];
```

```
# ------------------------- root_MCNFEdge_Net.mod ----------------------
set VERTS, EDGES, COMM;
param edge_source{EDGES}, edge_target{EDGES};
param comm_source{COMM}, comm_target{COMM}, comm_demand{COMM};
param b{k in COMM, i in VERTS} :=
  if  (comm_source[k]==i) then comm_demand[k] else
    if (comm_target[k]==i) then -comm_demand[k] else 0;

set EDGES_SUB within EDGES;
set EDGEDIFF{e in EDGES} = EDGES diff {e};  # local EDGES still present

set COMM_SUB within COMM;
var MCNFEdge_Net_Flow{e in EDGES_SUB, k in COMM_SUB, EDGEDIFF[e]} >= 0;
# flow into vertex - flow out of vertex equals demand
subject to MCNFEdge_Net_FlowBalance{e in EDGES_SUB, k in COMM_SUB, i in VERTS}:
      sum{j in EDGEDIFF[e]:edge_target[j]==i} MCNFEdge_Net_Flow[e,k,j]
      - sum{j in EDGEDIFF[a]:edge_source[j]==i} MCNFEdge_Net_Flow[e,k,j] = b[k,i];
  }
}
```

Fig. 12 Generated AMPL model files for the root MSND model and MCNF submodels

the needs of all different conceivable structure-exploiting solvers. A full discussion of the `ExpandedTreeNode` class can be found in the SML documentation.

6 Conclusions and future work

In this paper we have presented a new structure-conveying algebraic modelling language (SML) for mathematical and stochastic programming. It supplements the standard AMPL modelling language with extensions that allow models to be constructed from sub-models in an elegant and natural way. There is a noticeable reduction in the complexity of the resulting model, especially in the indexing of variables, and the

SML formulation exposes the inherent structure of the problem. The strength of the approach lies in the fact that this block structure is passed on to solvers that can exploit it [4,18,28]. SML supports the modelling of stochastic programs with recourse based on a stage-wise description. Information on the depth and shape of the associated scenario tree is provided as data, thus achieving a complete separation between model and data.

An SML parser for linear, quadratic and stochastic programming problems is already available. Its extension to nonlinear programming problem and the support for parallel model generation are work in progress.

One major advantage of solvers that exploit structure is the ease with which they can be adapted to solve sub-problems in parallel. The inherent parallelism in the problem is also known to the AML processor and, to keep this advantage, this information should be used to parallelize the problem generation process itself [21]. There are two issues that make this non-trivial. Firstly, distributing sub-problems evenly amongst the available processors could lead to load balancing problems, and so some understanding of the relative size or difficulty of each sub-problem is required. Secondly, the distribution of sub-problems should be guided by the solver to enable node-specific solver files to be created locally on the correct processor and avoid excessive data transfers. This will require a more sophisticated SML-solver interface than that described above.

Even without a parallel sub-model generation process, we believe the design of our structure-conveying modelling language is scalable to truly large problems.

This feature will become of major importance in the future, as optimisation models keep growing in size: although huge optimization problems are solvable by modern parallel solvers, such problems are now frequently beyond the reach of current modelling languages.

7 Code availability

The implementation of SML we describe has been released under an open source license and is distributed along with a limited version of the solver OOPS (proprietary license). Full details of the packages and their license conditions are available at

http://www.maths.ed.ac.uk/ERGO/sml/

Acknowledgments We thank Robert Fourer, the referees and the technical editor for their comments, which led to a more precise presentation.

References

1. Benders, J.F.: Partitioning procedures for solving mixed-variables programming problems. Numerische Mathematik **4**, 238–252 (1962)
2. Birge, J., Dempster, M., Gassmann, H., Gunn, E., King, A., Wallace, S.: A standard input format for multiperiod stochastic linear programs. Comm. Algorithms Newsl. **17**, 1–19 (1987)
3. Bisschop, J., Entriken, R.: AIMMS the modeling system. Paragon Decis. Technol. (1993)
4. Blomvall, J., Lindberg, P.O.: A Riccati-based primal interior point solver for multistage stochastic programming. Eur. J. Oper. Res. **143**, 452–461 (2002)

5. Brooke, A., Kendrick, D., Meeraus, A.: GAMS: A User's Guide. The Scientific Press, Redwood City, CA (1992)
6. Buchanan, C., McKinnon, K., Skondras, G.: The recursive definition of stochastic linear programming problems within an algebraic modeling language. Ann. Oper. Res. **104**, 15–32 (2001)
7. Colombani, Y., Heipcke, S.: Mosel: an extensible environment for modeling and programming solutions. In: Jussien, N., Laburthe, F. (eds.) Proceedings of the Fourth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR '02), pp. 277–290. Le Croisic, France (2002)
8. Dantzig, G.B., Wolfe, P.: The decomposition algorithm for linear programming. Econometrica **29**, 767–778 (1961)
9. Ferris, M.C., Horn, D.J.: Partitioning mathematical programs for parallel solution. Math. Programm. **80**, 35–62 (1998)
10. Fourer, R.: Proposed new AMPL features: stochastic programming extensions. http://www.ampl.com/NEW/FUTURE/stoch.html. Accessed 10 February 2009
11. Fourer, R., Gay, D.M.: Conveying problem structure from an algebraic modeling language to optimization algorithms. In: Laguna, M., Velarde, J.G. (eds.) Computing Tools for Modeling, Optimization and Simulation: Interfaces in Computer Science and Operations Research, pp. 75–89. Kluwer, Dordrecht (2000)
12. Fourer, R., Lopes, L.: StAMPL: A filtration-oriented modeling tool for stochastic recourse problems. INFORMS Journal on Computing **21**, 242–256 (2009)
13. Fourer, R., Gay, D., Kernighan, B.W.: AMPL: A Modeling Language for Mathematical Programming. The Scientific Press, San Francisco, CA (1993)
14. Fragnière, E., Gondzio, J.: Optimization modeling languages. In: Pardalos, P., Resende, M. Handbook of Applied Optimization, pp. 993–1007. Oxford University Press, New York (2002)
15. Fragnière, E., Gondzio, J., Sarkissian, R., Vial, J.-P.: Structure exploiting tool in algebraic modeling languages. Management Science **46**, 1145–1158 (2000)
16. Gay, D.M.: Hooking your solver to AMPL, tech. rep., Bell Laboratories, Murray Hill, NJ, (1993) (revised 1994, 1997)
17. Gondzio, J., Grothey, A.: Direct solution of linear systems of size 10^9 arising in optimization with interior point methods. In: Wyrzykowski, R. Parallel Processing and Applied Mathematics, pp. 513–525. Springer, Berlin vol. 3911 of Lecture Notes in Computer Science (2006)
18. Gondzio, J., Grothey, A.: Parallel interior point solver for structured quadratic programs: Application to financial planning problems. Ann. Oper. Res. **152**, 319–339 (2007)
19. Gondzio, J., Grothey, A.: Exploiting structure in parallel implementation of interior point methods for optimization. Comput. Manage. Sci. **6**, 135–160 (2009)
20. Gondzio, J., Sarkissian, R.: Parallel interior point solver for structured linear programs. Math. Programm. **96**, 561–584 (2003)
21. Grothey, A., Hogg, J., Woodsend, K., Colombo, M., Gondzio, J.: A structure-conveying parallelisable modelling language for mathematical programming. In: Čiegis, R., Henty, D., Kågström, B., Žilinskas, J. (eds.) Parallel Scientific Computing and Optimization: Advances and Applications, pp. 147–158. Springer, Berlin vol. 27 of pringer Optimization and Its Applications (2009)
22. Hochreiter, R.: Simplified stage-based modeling of multi-stage stochastic programming problems. In: Conference Talk: SPXI, Vienna, Austria, August (2007)
23. Kuip, C.A.C.: Algebraic languages for mathematical programming. Eur. J. Oper. Res. **67**, 25–51 (1993)
24. Markowitz, H.: Portfolio selection. J. Finance **7**(1), 77–91 (1952)
25. Murtagh, B.: Advanced Linear Programming: Computation and Practice. McGraw-Hill, New York (1981)
26. Refalo, P.: Linear formulation of constraint programming models and hybrid solvers. In: Dechter, R. (ed.) Principles and Practice of Constraint Programming—CP 2000, pp. 369–383. Springer, Berlin/Heidelberg vol. 1894/2000 of Lecture Notes in Computer Science (2000)
27. Sengupta, P.: MILANO, an object-oriented algebraic modeling system. Available from Simulation Sciences Inc. website: http://www.simsci-esscor.com/NR/rdonlyres/C0061CFF-7F7A-4526-8710-D4362C6DCD36/0/102797.pdf
28. Steinbach, M.: Hierarchical sparsity in multistage convex stochastic programs. In: Uryasev, S., Pardalos, P.M. (eds.) Stochastic Optimization: Algorithms and Applications, pp. 363–388. Kluwer, Dordrecht (2000)

29. Valente, C., Mitra, G., Sadki, M., Fourer, R.: Extending algebraic modelling languages for stochastic programming. INFORMS J. Comput. **21**, 107–122 (2009)
30. Van Hentenryck, P.: The OPL Optimization Programming Language. MIT Press, Cambridge (1999)
31. Wallace, S.W., Ziemba , W.T. (eds.): Applications of Stochastic Programming. SIAM, Philadelphia (2005)