

Structure-Conveying Modelling Language

SML User's Guide

Marco Colombo Andreas Grothey Jonathan Hogg
Kristian Woodsend Jacek Gondzio

School of Mathematics and Maxwell Institute,
University of Edinburgh, UK

<http://www.maths.ed.ac.uk/ERGO/sml/>
sml-support@maths.ed.ac.uk

February 18, 2011

Table of Contents

Introduction	3
1 Installation	4
1.1 Third-party software	4
1.2 Configuration Options	5
1.3 Running SML	5
2 Structured Problems	6
2.1 Example Problem 1: Survivable Network Design	6
2.1.1 Standard modelling language formulation	7
2.2 Example Problem 2: Asset and Liability Management	8
2.3 Identifying the structure	9
2.3.1 MSND Example	9
2.4 Stochastic Structure	10
2.4.1 ALM Example	10
3 SML Keywords	11
3.1 Blocks	11
3.2 Stochastic Blocks and Variables	11
4 Advanced Topics	13
4.1 Solver Interface	13
4.1.1 Data structure	13
4.1.2 Basic Use	13
4.1.3 API	14
4.1.4 Returning the solution	15
4.2 Implementation	16
Bibliography	18

Introduction

Foreword

Algebraic modelling languages 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 AIMMS [1], GAMS [2], Xpress-Mosel [3] and AMPL [4].

SML (Structured Modelling Language) attempts build on these traditional approaches while encapsulating the **structure** of the problem. As such it encapsulates the AMPL syntax in an object oriented fashion. We believe that object-oriented modelling will eventually confer the following benefits:

Clarity of modelling, reducing the apparent complexity of models

Reusability of model parts, reducing errors and allowing more complex systems to be modelled.
(Planned for later releases)

Speed of solution, encouraging a structured model which can be exploited by transparently passing structural information to the solver.

References

It would be greatly appreciated if users would cite the following references in work that uses SML:

- MARCO COLOMBO, ANDREAS GROTHEY, JONATHAN HOGG, KRISTIAN WOODSEND AND JACEK GONDZIO, *A Structure-Conveying Modelling Language for Mathematical and Stochastic Programming*, Technical Report ERGO 09-003, School of Mathematics. Accepted for publication in *Mathematical Programming Computation*.
- ANDREAS GROTHEY, JONATHAN HOGG, KRISTIAN WOODSEND, MARCO COLOMBO AND JACEK GONDZIO, *A Structure Conveying Parallelizable Modelling Language for Mathematical Programming*, Springer Optimization and Its Applications Vol. 27: Parallel Scientific Computing and Optimization ed: R. Čiegis, D. Henty, B. Kågström, and J. Žilinska. 2009.

Reproduction

This user's guide is licensed as part of SML under the Lesser GNU Public License (LGPL) version 3. You can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, using version 3 of the License.

You should have received a copy of the GNU Lesser General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

1 Installation

SML uses the standard GNU autotools install process. This can be summarised by the following steps:

1. `./configure --with-ampl=/path/to/amplsolver`
2. `make`
3. `make install`

However there are number of options and pieces of third party software you will require.

1.1 Third-party software

As SML is currently implemented as a pre- and post-processor to AMPL you will need an installed version of AMPL. You will also need the Ampsolver library available from Netlib. These two components are sufficient to build and use SML through the `smlmps` interface. Other two interfaces (`smlloops` and `smlcplex`) can be built if additional software libraries are available.

The third-party software you will need is summarised in Table 1.1.

Software	Required?	Free?	URL
AMPL	Yes	No	http://www.ampl.com/vendors.html
<code>amplsolver.a</code>	Yes	Yes	http://www.netlib.org/ampl/solvers/
OOPS	No	Limited	http://www.maths.ed.ac.uk/gondzio/parallel/solver.html
BLAS	for OOPS	Yes	http://www.netlib.org/blas/
LAPACK	for OOPS	Yes	http://www.netlib.org/lapack/
METIS	No	Yes	http://glaros.dtc.umn.edu/gkhome/views/metis
Cplex	No	No	http://ilog.com/products/cplex/

Table 1.1: Third-party software and where to get it

OOPS is the Object-Oriented Parallel Solver by Gondzio and Grothey [5].

If you wish to use the OOPS solver then you will also need the OOPS library. A serial version, limited to 5000 constraints, is distributed with SML under a closed-source license and may only be linked with the LGPL SML library. See `ThirdParty/oops/COPYING` for full terms. The version supplied is compiled using GNU `g++` and `gfortran` on a 32-bit Linux machine; if your setup differs then mileage may vary.

If you have a full version of OOPS (contact Andreas Grothey <A.Grothey@ed.ac.uk> for further information) you may use this instead, though you may also need the METIS library. OOPS also requires LAPACK and BLAS routines; we advise that you use the ones supplied by your computer vendor, however there are freely available implementations available from the Netlib archive.

1.2 Configuration Options

The following configuration options are available:

<code>--with-ampl=path</code>	Absolute path to directory containing libamplsolver.a
<code>--with-oops=path</code>	Absolute path to OOPSHOME directory
<code>--with-cplex=path</code>	Absolute path to the Cplex installation
<code>--with-blas=library</code>	BLAS library to use
<code>--with-lapack=library</code>	LAPACK library to use
<code>--with-metis=path</code>	Absolute path to METIS directory
<code>--with-mps=[yes no]</code>	Enable/disable MPS interface

Running the command:

```
./configure --help
```

will display a more comprehensive summary.

1.3 Running SML

You can find some small examples of problems modelled in SML in the `tests` directory. There are two files for each problem: the model file and the data file. This forces the user to write a model that is general enough so that it can be reused with different data sets.

The `smlmps` interface parses the model and data files to produce an MPS file, which can then be used as input file to virtually any solver for solution. For example, this can be accomplished with the following command:

```
smlmps alm.mod alm.dat alm.mps
```

To see how an interface can be called and what options it supports, call it with the “`--help`” option, such as:

```
smlmps --help
```

2 Structured Problems

A typical mathematical programming problem can be written in the form

$$\min_x f(x) \quad \text{s.t.} \quad Ax = b, \quad x \geq 0 \quad (2.1)$$

If the matrix A is structured then this can be exploited by modern solvers to solve the problem faster.

This structure can take a number of forms, for example if the problem involves a network then the node-arc incidence matrix may be repeated multiple times within the matrix A , or if the problem is stochastic in nature then the scenario tree can be exploited.

Throughout this guide we will consider two structured problems:

MSND Survivable Network Design

ALM Asset and Liability Management

2.1 Example Problem 1: Survivable Network Design

A network is described by sets \mathcal{V} of nodes and \mathcal{E} of (directed) arcs and a base capacity C_j for every arc $j \in \mathcal{E}$. 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 arcs of the shared-capacity network. Each commodity is described as the triplet (s_k, t_k, d_k) consisting of start node s_k , terminal node t_k and amount to be shipped d_k . A feasible flow $x_k = (x_{k,j})_{j \in \mathcal{E}}$ for the k -th 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 node-arc incidence matrix, and $b_k = (b_{k,i})_{i \in \mathcal{V}}$ is the demand vector for the k -th commodity, with the following conventions:

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

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 arc or node 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 arcs or nodes is removed. Note that the subproblems are not completely independent, as they are linked by the common spare capacities S_j .

Let $A^{(a,l)}$, $l \in \mathcal{E}$, be the node-arc incidence matrix in which the column corresponding to the l -th arc is set to zero. Then any vector of flows $x_k^{(a,l)} \geq 0$ satisfying

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

gives a feasible flow to route the k -th commodity through the arc-reduced network. Similarly let $A^{(n,i)}$, $i \in \mathcal{V}$, be the node-arc incidence matrix in which the row corresponding to the i -th nodes and the columns corresponding to arcs incident to this node are set to zero. Then any vector of flows $x_k^{(n,i)} \geq 0$ satisfying

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

gives a feasible flow to route the k -th commodity through the node-reduced network. As the network is capacity-limited, however, each arc $j \in \mathcal{E}$ can carry at most $C_j + S_j$ units of flow. The complete formulation of the MSND problem is as follows:

$$\begin{aligned}
\min \quad & \sum_{j \in \mathcal{E}} c_j S_j \\
\text{s.t.} \quad & A^{(a,l)} x_k^{(a,l)} = b_k \quad \forall k \in \mathcal{C}, l \in \mathcal{E} \\
& A^{(n,i)} x_k^{(n,i)} = b_k \quad \forall k \in \mathcal{C}, i \in \mathcal{V} \\
& \sum_{k \in \mathcal{C}} x_{k,j}^{(a,l)} \leq C_j + S_j \quad \forall j \in \mathcal{E}, l \in \mathcal{E} \\
& \sum_{k \in \mathcal{C}} x_{k,j}^{(n,i)} \leq C_j + S_j \quad \forall j \in \mathcal{E}, i \in \mathcal{V} \\
& x, s \geq 0
\end{aligned} \tag{2.2}$$

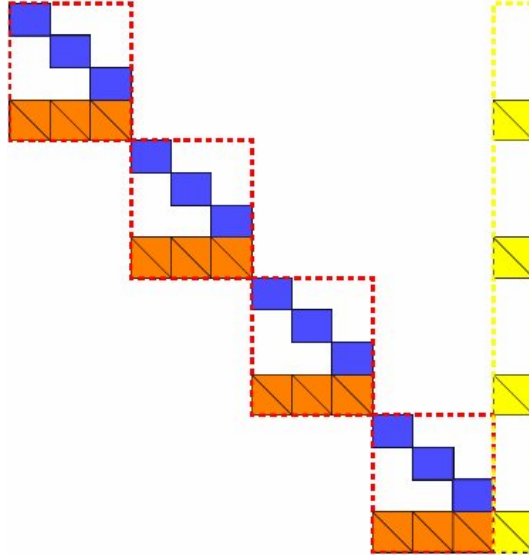


Figure 2.1: Structure of the constraint matrix for the MSND problem.

The constraint matrix of this problem has the form shown in Figure 2.1. The basic building blocks are the node-arc 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) arc and node. The common capacity variables (light grey blocks) act as linking columns. While the nested structure of the problem is obvious in Figure 2.1, it cannot be easily appreciated from the mathematical formulation (2.2).

2.1.1 Standard modelling language formulation

Problem (2.2) can be represented in an algebraic modelling language as shown below, 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.

```

set NODES, ARCS, COMM;
param cost{ARCS}, basecap{ARCS};
param arc_source{ARCS}, arc_target{ARCS};
param comm_source{COMM}, comm_target{COMM}, comm_demand{COMM};
param b{k in COMM, i in NODES} :=
    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 arc/node, then commodity, then arc of flow
var Flow{ARCS union NODES, COMM, ARCS} >= 0;
var sparecap{ARCS} >= 0;

# flow into node - flow out of node equals demand
subject to FlowBalanceMissingArcs{a in ARCS, k in COMM, i in NODES}:
    sum{j in ARCS:j~=a, arc_target[j]==i} Flow[a,k,j]
    - sum{j in ARCS:j~=a, arc_source[j]==i} Flow[a,k,j] = b[k,i];

subject to FlowBalanceMissingNodes{n in NODES, k in COMM, i in NODES diff {n}}:
    sum{j in ARCS:arc_target[j]==i, arc_source[j]~=n} Flow[n,k,j]
    - sum{j in ARCS:arc_source[j]==i, arc_target[j]~=n} Flow[n,k,j] = b[k,i];

subject to CapacityMissingArcs{a in ARCS union NODES, j in ARCS}:
    sum{k in COMM} Flow[a,k,j] <= basecap[j] + sparecap[j];

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

```

2.2 Example Problem 2: Asset and Liability Management

As an example of a stochastic programming problem we use an asset and liability management problem. 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 , $x_{i,j}^h$ is the amount of asset j held at node i , $x_{i,j}^b$ and $x_{i,j}^s$ 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 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 objective maximises a linear combination of the expectation of the final portfolio value (denoted by μ) and its variance, with risk-aversion parameter λ :

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

The complete model is the following:

$$\begin{aligned}
\max_{x, \mu \geq 0} \quad & \mu - \rho \left[\sum_{i \in \mathcal{L}_T} p_i \left[(1-c) \sum_{j \in \mathcal{A}} v_j x_{i,j}^h \right]^2 - \mu^2 \right] \\
\text{s.t.} \quad & x_{i,j}^h = (1 + r_{i,j}) x_{\pi(i),j}^h + x_{i,j}^b - x_{i,j}^s, \quad \forall i \neq 0, j \in \mathcal{A} \quad (2.3) \\
& \sum_{j \in \mathcal{A}} (1-c) v_j x_{i,j}^s = l_{\tau(i)} + \sum_{j \in \mathcal{A}} (1+c) v_j x_{i,j}^b, \quad \forall i \neq 0 \\
& \sum_{j \in \mathcal{A}} (1+c) v_j x_{0,j}^b = b \\
& (1-c) \sum_{i \in \mathcal{L}_T} p_i \sum_{j \in \mathcal{A}} v_j x_{i,j}^h = \mu
\end{aligned}$$

2.3 Identifying the structure

How do we identify the structure? Generally if you are building your model out of repeated components you can easily obtain structure - it may be that this structure would be indexed over in the traditional Algebraic Modelling Language representation - for example the indexing over ARCS and NODES in the MSND example, and the indexing over the scenario tree in the stochastic ALM example.

If this indexing is pulled out into a series of related blocks then the problem becomes similar. In the next example we consider the structured representation of the MSND model.

2.3.1 MSND Example

Using the block keyword, the previous MSND model can be rewritten as:

```
set NODES, ARCS, COMM;
param cost{ARCS}, basecap{ARCS};
param arc_source{ARCS}, arc_target{ARCS};
param comm_source{COMM}, comm_target{COMM}, comm_demand{COMM};
param b{k in COMM, i in NODES} :=
  if (comm_source[k]==i) then comm_demand[k] else
  if (comm_target[k]==i) then -comm_demand[k] else 0;

var sparecap{ARCS} >= 0;

block MCNFArcs{a in ARCS}: {
  set ARCSDIFF = ARCS diff {a}; # local ARCS still present

  block Net{k in COMM}: {
    var Flow{ARCSDIFF} >= 0;
    # flow into node - flow out of node equals demand
    subject to FlowBalance{i in NODES}:
      sum{j in ARCSDIFF:arc_target[j]==i} Flow[j]
      - sum{j in ARCSDIFF:arc_source[j]==i} Flow[j] = b[k,i];
  }

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

block MCNFNodes{n in Nodes}: {
  set NODESDIFF = NODES diff {n}; # local NODES still present
  set ARCSDIFF = {i in ARCS: arc_source[i]~n, arc_target[i]~n};

  block Net{k in COMM}: {
    var Flow{ARCS} >= 0;
    # flow into node - flow out of node equals demand
    subject to FlowBalance{i in NODESDIFF}:
      sum{j in ARCSDIFF:arc_target[j]==i} Flow[j]
      - sum{j in ARCSDIFF:arc_source[j]==i} Flow[j] = b[k,i];
  }

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

```

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

```

2.4 Stochastic Structure

As stochastic programming is a common source of such structured problems, and the special conventions used to represent stochastic entities, SML includes some special conventions for handling stochastic problems. Our next example shows these in action.

2.4.1 ALM Example

```

param Budget, T, Tc, Rho;
set ASSETS, NODES, STAGESSET := 0..T;
param PARENT{NODES} symbolic, PROBS{NODES};
param Value{ASSETS};
var mu;

block alm stochastic using (NODES, PARENT, PROBS, STAGESSET):{

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

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

  stage {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-Tc) * sum{j in ASSETS} Value[j]*xs[j] =
        Liability + (1+tc)*sum{j in ASSETS} Value[j]*xb[j];
  }

  stage {T}: {
    var wealth := (1-tc) * sum{j in ASSETS} Value[j]*xh[j];
    subject to ExpPortfolioValue:
      Exp(wealth) = mu;
    maximize objFunc: mu - Rho * ((wealth*wealth) - mu*mu )
  }
}

```

3 SML Keywords

In order to facilitate Structured Modelling we have introduced several new keywords and structures into the AMPL language. These keywords are the following reserved words which may not be used for component names:

<code>ancestor</code>	<code>Exp</code>	<code>stochastic</code>
<code>block</code>	<code>node</code>	<code>suffix</code>
<code>deterministic</code>	<code>stages</code>	<code>using</code>

We also note that while AMPL allows the redefinition of certain keywords (for example `from` and `prod`) rendering their erstwhile meanings unavailable, `sml` does not permit this for the following reasons:

1. It complicates the coding required to handle them.
2. We believe it makes the model more error prone and hard to read.

3.1 Blocks

A block is defined as follows:

```
block nameofblock [{j in set_expr}] : {  
  [statements]  
}
```

Between the curly braces `{}`, any number of `set`, `param`, `subject to`, `var`, `minimize` or indeed further `block` definitions can be placed. The square brackets `[]` are not part of the language and are merely used to indicate optional parts of the expression. 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 introduces the notion of scope, similar to that found in the programming language C. Within a block all references to objects visible to the containing context are visible *unless* there is an object with the same name defined within the block.

3.2 Stochastic Blocks and Variables

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

```
block nameofblock [{k in set_expr] stochastic using ({i in NODES, PROB, PARENT,  
  [j in STAGESET]) : {  
  [stochastic statements]  
}
```

where the expressions `NODES`, `PROB`, `PARENT` and `STAGESET` conform to the following relations:

```
set NODES;
param PROB{NODES};
param PARENT{NODES} symbolic, within (NODES union {none});
set STAGESET ordered;
```

The `stochastic` statements include both normal AMPL and `block` statements, but may also include the special `stages` statement:

```
stages set_expr: {
  [stochastic statements]
}
```

that declares variables and constraints which apply only to the specific stages in the supplied set expression.

Currently, no check is performed on the validity of the values indicated in the `PROB` set: the user has to be careful to indicate positive values such that for all stages their sum is 1.

The scenario tree is defined by the set of `NODES` and their `PARENT` relationship. The probability associated with a node is the conditional probability of reaching that node given that its parent is reached. The set expression `STAGESET` is used only to provide labels for the stages should they be required. It is assumed that the number of levels thus defined in the tree matches the cardinality of the `STAGESET` set.

Entities defined within the stochastic block are either *stochastic* and are repeated for nodes in their *stageset*, or are *deterministic* and are repeated only once for each stage. By default entities are stochastic and their stageset is the full set `STAGESET`. Entities within a `stages` block are by default stochastic with their stageset determined by the `stages` set expression. The nature of an entity may be changed from the default through the use of the special attributes `deterministic` and `stages set_expr` in their declaration.

The dummy indices of the `NODES` and `STAGESET` set expressions allow the reference of the particular node or stage within the block through the normal indexing mechanism.

Relations between different stages can be expressed by using the `ancestor(i)` function that allows to reference the *i*-th ancestor stage. Variables of the ancestor can be referenced using the normal SML syntax, `ancestor(i).x`.

Expectations may also be handled through a special new syntax. Within a stochastic block the expression `Exp(expr)` is equivalent to the expression

$$\text{sum}\{\text{nd in NODES: nd in } \textit{currentstage}\} \text{ PROBS}[\text{nd}] * \text{expr}[\text{nd}]$$

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

4 Advanced Topics

4.1 Solver Interface

If you wish to hook a solver other than OOPS to SML then you will need to use the interface described in this section. We first describe the data representation used by the interface before describing the available function calls. The implementer is advised to browse the source code for the MPS interface for an example implementation.

4.1.1 Data structure

Internally, SML describes a problem as a nested bordered block diagonal form. The data structure used to represent this assumes the following:

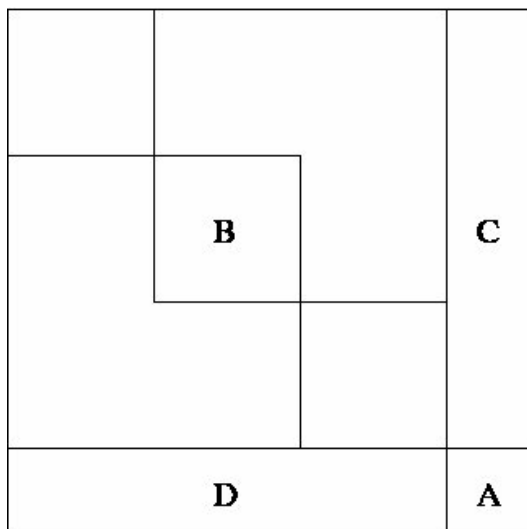
1. A block has a zero or more constraints of variables with an associated Jacobian matrix, labelled A in Figure 4.1.
2. Further, associated variables may contribute to the objective function.
3. A block has zero or more additional blocks nested within it, these are represented by a list of pointers to the contained blocks.
4. A contained block has a similar internal structure, labelled B, that has no bearing on the constraints of variables associated with this block.
5. The constrained block may contain constraints on the variables associated with this block, with a Jacobian labelled C.
6. This block may place constraints upon variables associated with the contained block, yielding the Jacobian labelled D.
7. Any or all of the Jacobians A, B, C or D may be identically zero.

The list of pointers to contained blocks yields a tree structure which we refer to as the *Expanded Model Tree*. This representation of the structured model is natural for a hierarchical linear algebra solver such as OOPS, but should also provide sufficient access for other designs of structure exploiting solver such as those which use decomposition approaches.

4.1.2 Basic Use

The model currently used by SML is that it is called by the solver and passed the names of the model and data files, it will then return a pointer to an Expanded Model Tree represented by a `ExpandedModelInterface` object which the solver is then free to query. Finally, the solver should delete the `ExpandedModelInterface` object to free the memory.

Figure 4.1: Structure of the constraint matrix for the MSND problem.



We recommend that solvers parse the tree in either one or two passes. The initial parse may extract information on the dimension and relative density of the tree, setting up data structures are required, while the second pass will determine the actual numeric values.

4.1.3 API

Acquiring the `ExpandedModelInterface` object

The solver should use the include statement

```
#include "sml.h"
```

which will provide access to all relevant definitions. The solver should then call the function:

```
ExpandedModelInterface* sml_generate(const std::string modelfilename,
    const std::string datafilename, const bool debug);
```

to retrieve a `ExpandedModelInterface` object which can then be queried. If the argument `debug` is `true` then additional debugging information will be generated.

Traversing the Tree

The `ExpandedModelInterface` object returned after parsing the model and data files offers two iterators for moving through the tree, and further allows direct access to the list of children:

- The `ExpandedModelInterface::ancestor_iterator` allows the iteration up through the tree to the root. The function `ExpandedModelInterface::abegin()` returns an iterator which points to the parent of the node it is called on. The function `ExpandedModelInterface::aend()` return an iterator which represents the top of the tree, above the root.

- The `ExpandedModelInterface::child_iterator` allows the iteration over all children of a node in a depth-first order, with the node itself being last. The function `ExpandedModelInterface::cbegin()` returns an iterator pointing to the left-most leaf in the subtree routed at the calling node. The function `ExpandedModelInterface::cend()` returns an iterator representing the end of the child iteration sequence.
- The component `std::vector<ExpandedModelInterface*> ExpandedModelInterface::children` allows the implementer to use their own iteration scheme if the above are not sufficient.

We note that (non-)equivalence of pointers is not a sufficient condition to differentiate nodes, and the function `ExpandedModelInterface::getName()` should be used instead to compare the names of the blocks (which are guaranteed to be unique).

Accessing the Data

Each block essentially allows the query of all data in its row block. The following functions are available:

<code>int getNLocalVars()</code>	Number of variables local to block
<code>const std::list<std::string>& getLocalVarNames()</code>	List of variable names local to block
<code>int getNLocalCons()</code>	Number of constraints local to block
<code>const std::list<std::string>& getLocalConNames()</code>	List of constraint names local to block
<code>int getNzJacobianOfIntersection(ExpandedModelInterface *emcol)</code>	Number of non-zeros in submatrix in block column <code>emcol</code> and this block row
<code>void getJacobianOfIntersection(ExpandedModelInterface *emcol, *colbeg, *collen, *rownbs, *el)</code>	Entries of submatrix in block column <code>emcol</code> and this block row, in CSC format (see below)
<code>void getRowLowBounds(double *elts)</code>	Constraint lower bounds for block.
<code>void getRowUpBounds(double *elts)</code>	Constraint upper bounds for block.
<code>void getColLowBounds(double *elts)</code>	Variable lower bounds for block.
<code>void getColUpBounds(double *elts)</code>	Variable upper bounds for block.
<code>void getObjGradient(double *elts)</code>	Objective coefficients for block.

The Compressed Sparse Column (CSC) format used by `getJacobianOfIntersection()` describes for each column only the non-zero entries within it, and for each entry gives a row number and value pair. On entry the parameters should point to areas of memory corresponding to the following arrays:

```
int colbeg[n+1], int collen[n], int rownbs[nz], double el[nz]
```

where `n` is the number of variables in this model, and `nz` is the number returned by the corresponding call to `getNzJacobianOfIntersection()`.

On return, `colbeg[i]` is a pointer to the beginning of column `i` within the arrays `rownbs[]` and `el[]`, `collen[i]` is the number of entries in that column — which may be zero. The values `rownbs[j]` and `el[j]` represent a row number-value pair.

4.1.4 Returning the solution

You may return the solution to SML and have it write out a file containing these solutions. The following functions are available for each `ExpandedModelInterface` block to return the primal and dual solutions associated with each variable and constraint within the block:

<code>void setPrimalSolColumns(double *elts)</code>	Set the primal solution vector for variables in this block.
<code>void setDualSolColumns(double *elts)</code>	Set the dual solution vector for variables in this block.
<code>void setPrimalSolRows(double *elts)</code>	Set the primal solution vector for constraints in this block.
<code>void setDualSolRows(double *elts)</code>	Set the dual solution vector for constraints in this block.

The complete solution may be output recursively for a block (ie the solution associated with this block and all its descendants) by calling the function

```
void outputSolution(std::ostream &out [, int indent] )
```

which will write the solution to the output stream `out`. If the optional argument `indent` is supplied that the solution will be indented by a number of spaces equal to `indent`. If `indent` is not present, then no indent will be used.

4.2 Implementation

SML is implemented in the object-oriented C++ language as a pre- and post-processor to AMPL. The SML 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 easily generated from the SML model file by generic text substitutions. Figure 4.2 shows these AMPL submodel files for the MSND problem formulation.

The AMPL model is separated into the appropriate submodels for every node of the prototype 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 `ARCS.SUB` to a suitable subset of `ARCS` (leveraging AMPL's scripting commands), the model for any node on the expanded tree can be generated from the corresponding submodel `*.mod` file shown in Figure 4.2.

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 the needs of all different conceivable structure-exploiting solvers.

```

%----- root.mod -----
set ARCS;
param cost{ARCS};

var sparecap{ARCS} >= 0;

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

```

```

% ----- root_MCNFArcs.mod -----
set ARCS, COMM;
param basecap{ARCS};

var sparecap{ARCS} >= 0;

set ARCS_SUB within ARCS;
set ARCSDIFF{a in ARCS_SUB} = ARCS diff {a}; # local ARCS still present

var MCNFArcs_Net_Flow{a in ARCS_SUB, ARCSDIFF[a], k in COMM} >= 0;
subject to Capacity{a in ARCS_SUB, j in ARCSDIFF[a]}:
    sum{k in COMM} MCNFArcs_Net_Flow[a,k,j] <= basecap[j] + sparecap[j];

```

```

% ----- root_MCNFArcs_Net.mod -----
set NODES, ARCS, COMM;
param arc_source{ARCS}, arc_target{ARCS};
param comm_source{COMM}, comm_target{COMM}, comm_demand{COMM};
param b{k in COMM, i in NODES} :=
    if (comm_source[k]==i) then comm_demand[k] else
    if (comm_target[k]==i) then -comm_demand[k] else 0;

set ARCS_SUB within ARCS;
set ARCSDIFF{a in ARCS} = ARCS diff {a}; # local ARCS still present

set COMM_SUB within COMM;
var MCNFArcs_Net_Flow{a in ARCS_SUB, k in COMM_SUB, ARCSDIFF[a]} >= 0;
# flow into node - flow out of node equals demand
subject to MCNFArcs_Net_FlowBalance{a in ARCS_SUB, k in COMM_SUB, i in NODES}:
    sum{j in ARCSDIFF[a]:arc_target[j]==i} MCNFArcs_Net_Flow[a,k,j]
    - sum{j in ARCSDIFF[a]:arc_source[j]==i} MCNFArcs_Net_Flow[a,k,j] = b[k,i];
}
}

```

Figure 4.2: Generated AMPL model files for the root MSND model and MCNF submodels

Bibliography

- [1] J. BISSCHOP AND R. ENTRIKEN, *AIMMS The Modeling System*, Paragon Decision Technology, 1993.
- [2] A. BROOKE, D. KENDRICK, AND A. MEERAUS, *GAMS: A User's Guide*, The Scientific Press, Redwood City, California, 1992.
- [3] Y. COLOMBANI AND S. HEIPCKE, *Mosel: an extensible environment for modeling and programming solutions*, in Proceedings of the Fourth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'02), N. Jussien and F. Laburthe, eds., Le Croisic, France, Mar., 25–27 2002, pp. 277–290.
- [4] R. FOURER, D. GAY, AND B. W. KERNIGHAN, *AMPL: A Modeling Language for Mathematical Programming*, The Scientific Press, San Francisco, California, 1993.
- [5] J. GONDZIO AND R. SARKISSIAN, *Parallel interior point solver for structured linear programs*, Mathematical Programming, 96 (2003), pp. 561–584.