Extending Algebraic Modeling Languages to Support Algorithm Development for Solving Stochastic Programming Models

Suleyman Karabuk

University of Oklahoma, Industrial Engineering, karabuk@ou.edu

Abstract

An algebraic modeling language is a domain specific computer programming language for describing and solving mathematical programming models. We propose extending algebraic modeling languages so that solution algorithms that are based on iteratively manipulating, modifying, and solving a model are supported at a high abstraction level. We specifically focus on Stochastic Programming models with random parameters formulated as discrete scenarios, and mathematical decomposition algorithms, which are commonly applied to solve such models. We identify the necessary language constructs, and develop a design based on the open source modeling software APLEpy. The proposed design, although specifically addresses decomposition algorithms, proves useful for implementing heuristic solution algorithms as well. The object oriented nature of the design results in the algorithms, coded with the proposed extensions, to work, without any modification, with any other model that satisfy the assumptions of the initial model. This flexible and robust design helps inexperienced modelers to immediately apply an advanced solution algorithm, and experienced modelers to build sophisticated algorithms easily within the same development environment used to describe the model under consideration.

1 Introduction

We refer to the software tools that are used to describe a mathematical programming model, as a prelude to solving it, as algebraic modeling and programming software (AMPS). An AMPS product is a stand alone package that provides a specialized development environment and a domain specific (and vendor specific) programming language. The specialized language, which is referred to as an algebraic modeling language (AML), provides a high level interface to describe the elements of a model in a way that closely parallels its corresponding algebraic formulation.

As more involved modeling approaches, such as Stochastic Programming, start to become widely practised, the AMLs are extended to provide language constructs in dealing with the complexities of describing such model types. In this paper we focus on language features that support Stochastic Linear Programming (SLP) models with recourse. We assume that random input parameters are described in the form of discrete scenarios.

The SLP models we consider are expressed as large scale mathematical models with a special structure. The special modeling language constructs aim at describing this special structure in a simple and transparent manner to the modeler. However, these type of SLP models can rarely be solved as they are, due to their large size. Fortunately, there are well established exact and heuristic solution algorithms that take advantage of the special structure of a SLP model. These solution algorithms are based on defining subproblems by decomposing, reducing, modifying, restoring the original problem, and solving the subproblems in an iterative fashion. In particular, mathematical decomposition is the most commonly used approach used to solve a SLP model. The information needed to implement such algorithms is contained in the model description, however, an AMPS user has to gather and organize that information and describe the algorithm with a language that is not specifically designed for such use. Therefore, we submit that there is a need to provide AML extensions to support algorithm design to solve SLP models. This need is further evidenced by the fact

that most, if not all, AMPS packages come with example code that illustrates how to apply mathematical decomposition algorithms in their language.

1.1 Literature

A recent survey of AMPS products, and the current state of research in AMLs can be found in Fourer (2005), and Kallrath (2004) respectively. Research on SLP extensions for modeling languages have mostly focused on developing data structures and language constructs to support describing a SLP model (Gassmann and Ireland (1995); Fourer (1996); Gassmann and Ireland (1996); Gassmann (1998); Buchanan et al. (2001); Entriken (2001); Valente et al. (2001); Dominguez-Ballesteros et al. (2002); Domenica et al. (2004); Valente et al. (2005)). The AMPS vendors are also releasing their own implementation in their products. Some examples of packages with SLP modeling support are: Xpress-Mosel (Guéret et al. (2002)), SAMPL (Valente et al. (2005)), GAMS (Brooke et al. (1998)), and MPL (Kristjansson (2003)).

Lately, AMPS products are evolving to become an integrated development and solution environment, by providing options for applying high level solution algorithms. Of particular interest to SLP modeling are the SPInE (Valente et al. (2004)), and the MPL packages. These packages offer options to apply well known mathematical decomposition algorithms, automatically, to solve a SLP model. This kind of support is welcome, because SLP models are rarely solved as they are due to their large size.

However, it is not practical to provide every imaginable solution algorithm in a development environment. There are cases where a straightforward application of decomposition does not work and some tweaking is required. There is also some flexibility in determining the details of a mathematical decomposition algorithm, which would be difficult to capture by menu options. We believe that a better approach is to provide language extensions to describe high level solution algorithms, such as mathematical decomposition. That way, a large variety

of algorithms can be implemented by the modeler. The objective of this paper is to design modeling language constructs for algorithm development in solving large scale SLP models. These extensions would provide a higher abstraction layer for ease of development, while retaining enough flexibility to support implementing creative heuristic algorithms. We base the design on the open source modeling software APLEpy being developed by Karabuk (2005). The APLEpy package provides classes to describe a mathematical programming model in the Python programming language. Python is a general purpose, object oriented language, and as such it provides a robust and flexible foundation for our design.

As the first step of this overall research, we exclusively focus on supporting Lagrangian relaxation and L-shaped decomposition algorithms, which are commonly used for solving SLPs. The experience gained in this part would help make progress towards addressing general mathematical programming models, in a more general design framework.

In the next section, we formally describe the type of SLPs, and the decomposition algorithms we focus on. We also identify the requirements of the design in this section. This is followed by the description of the APLEpy environment in section 3. Next, we describe the proposed design in section 4 and demonstrate its use in APLEpy. Finally, we offer a conclusion in section 5.

2 Problem definition and scope

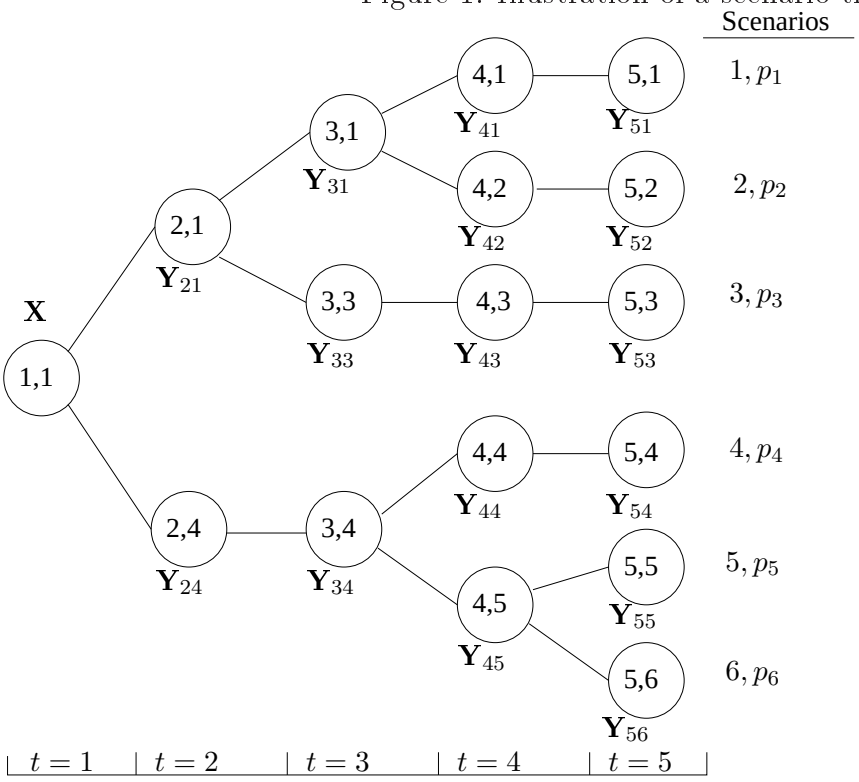
Stochastic programming approach explicitly incorporates uncertainty in the formulation of a mathematical programming model. This loosely defined purpose can be achieved in many forms resulting in different model structures as summarized by the taxonomy of SLPs in Gassmann and Ireland (1996). In this paper, we focus on recourse models with discrete scenarios. In a recourse model, the decision variables are partitioned into here-and-now decisions and wait-and-see decisions. As the names imply, the former set of decisions have to be made at the present time under uncertainty, and the latter decisions are deferred until

after uncertainty clears. The former type of decisions are made as a reaction to unfolding random events. These types of models have a high descriptive power and their special structure lends itself well to mathematical decomposition based solution algorithms. As a result, most SLP applications reported in the literature are based on recourse models.

Central to the formulation of a recourse model is a data structure referred to as a scenario tree, which shows stages in which random events unfold and decisions are made. Figure 1 shows an example scenario tree for a problem that spans five stages and includes six scenarios. A node in the tree is associated with outcome of an event that leads to a particular realization of the random parameters of the model; hence they are referred to as event nodes (nodes for short). The root node corresponds to the starting point of the planning horizon at which time no random event has cleared yet. As seen in the tree, nodes have unique labels (t,s) used as reference indices, which are utilized in the mathematical formulation of the problem. This labeling technique is due to Gassmann and Ireland (1995). A scenario is a complete path of nodes starting from the root node of the tree and terminating at a leaf node. It defines a sequence of events that make up one particular realization of the future through the planning horizon. Leaf nodes of the tree identify individual scenarios with an associated probability of occurrence, which in turn depends on the probabilities of the nodes that make up its path. Path probability of a node is represented by p_{ts} which we do not show in the figure for brevity. Note that p_{ts} values at leaf nodes are equal to p_s (scenario probability) values in figure 1. The decisions that are to be made independent of random outcomes (here-and-now decisions or stage-one decisions) are referred to by $\mathbf{X}$. The decisions that can be deferred until the outcome of an event becomes known (wait-and-see decisions or recourse decisions) are referred to by $\mathbf{Y}_{ts}$. The recourse variables are indexed by the associated event node (t,s).

The node labeling method we employ uses two indices t, and s: the first index t indicates the stage that the node is in, whereas, the second index s indicates the lowest numbered

Figure 1: Illustration of a scenario tree



scenario that shares the node. For example, consider scenario 2 (in figure 1) and its path of nodes [(1,1), (2,1), (3,1), (4,2),(5,2)]. It is the only scenario with path that goes through nodes (4,2) and (5,2) in stages 4 and 5 respectively, hence s=2 for both nodes. On the other hand, in t=3, both scenario 1 and scenario 2 share node (3,1), as a result s takes the value 1, which refers to the lowest numbered scenario. Similarly, in t=2, scenarios 1, 2, and 3 go through the same node (2,1), whose s index is set to 1. Also note that, a node takes its initial state from its parent node. This representation ensures that scenarios that share the same sequence of events also share the same decisions until they split. This is referred to as the nonanticipativity (NA) condition. A general formulation for a multistage SLP model with recourse is presented below.

SLP1

$$min \quad \mathbf{c}^x\mathbf{X} + \sum_{(t,s)\in E} p_{ts}\mathbf{c}^y_{ts}\mathbf{Y}_{ts}$$

subject to

$$\begin{aligned}
\mathbf{A}_t\mathbf{X} &= \mathbf{b}_t && t = 1 \\
\mathbf{B}_{ts}\mathbf{X} + \mathbf{A}_{ts}\mathbf{Y}_{ts} &= \mathbf{b}_{ts} && \forall (t,s) \in E | t = 2 \\
\mathbf{B}_{ts}\mathbf{Y}_{(t,prev(t,s))} + \mathbf{A}_{ts}\mathbf{Y}_{ts} &= \mathbf{b}_{ts} && \forall t \in T | t > 2, \forall (t,s) \in E
\end{aligned}$$

The formulation assumes that any of the model parameters (i.e., $\mathbf{c}$, $\mathbf{A}$, $\mathbf{b}$) can observe a random realization at a node. The function $prev(t,s)$ returns the scenario index of the predecessor of node (t,s). The notation E refers to the set of event nodes in the tree.

An alternative formulation, which is commonly exercised, is formed by describing each scenario problem independently and imposing the NA constraints explicitly. This formulation is shown below.

SLP2

$$min \quad \sum_{s\in S} p_s\mathbf{c}^x\mathbf{X}_s + \sum_{t=2}^{T}\sum_{s\in S} p_s\mathbf{c}^y_{ts}\mathbf{Y}_{ts}$$

subject to

$$\begin{aligned}
\mathbf{A}_t\mathbf{X}_s &= \mathbf{b}_t && t = 1, \forall s \in S \\
\mathbf{B}_{ts}\mathbf{X}_s + \mathbf{A}_{ts}\mathbf{Y}_{ts} &= \mathbf{b}_{ts} && t = 2, \forall s \in S \\
\mathbf{B}_{ts}\mathbf{Y}_{ts} + \mathbf{A}_{ts}\mathbf{Y}_{ts} &= \mathbf{b}_{ts} && t > 2, \forall s \in S \\
\mathbf{X}_s - \mathbf{X}_{\acute{s}} &= 0 && \forall s, \acute{s} \in S \quad (1) \\
\mathbf{Y}_{ts} - \mathbf{Y}_{t\acute{s}} &= 0 && \forall s, \acute{s} \in NA_{ts} \quad (2)
\end{aligned}$$

Note that the root node is also part of the scenario tree and it is shared by all scenarios. Therefore, stage one variables ($\mathbf{X}$) are also replicated for each scenario. However, constraints (1) ensure that every copy take the same value. Similarly, constraints (2) impose NA conditions on the rest of the nodes. For example, consider node (2,4) which is shared by scenarios 4, 5 and 6 in stage 2: constraints (2) include $\mathbf{Y}_{24} = \mathbf{Y}_{25} = \mathbf{Y}_{26}$.

2.1 Lagrangian relaxation solution

Formulation SLP2 is suited for use with Lagrangian relaxation based solution approaches. Algorithm 1 below outlines an application of the method to SLP2. Relaxing and dualizing the last two constraints of SLP2 results in the remaining formulation to decompose into independent scenario subproblems. The solution to the dualized problem, for any value of the dual multipliers, constitute a lower bound for the original problem. The algorithm iteratively searches for a set of dual multipliers (represented by λ in Algorithm 1) that maximizes the lower bound. There are different methods reported in the literature for both dualizing the constraints and searching the dual solution space. In most applications the dualized constraints are added as linear terms to the objective function. However, employing an augmented Lagrangian approach adds quadratic terms to the objective function (see Karabuk and Wu (2002) for an example application). The latter is proven to converge better at the expense of making the subproblems quadratic. Another advantage is that they require simple multiplier adjustment methods. The progressive hedging algorithm of Rockafellar and Wets (1991) is one well known approach based on augmented Lagrangian. Some of the dual search methods mentioned in the literature are, subgradient optimization, multiplier adjustment, and dual ascent techniques (Beasley (1993)). The subgradient optimization technique seems to be the most commonly used technique due to its empirical success. It requires the calculation of an upper bound at each iteration. This is done by applying a heuristic to convert the current infeasible solution to a feasible one. This upper bound, together with the current infeasibility amount of dualized constraints, is used to update the dual multipliers.

The Lagrangian relaxation algorithm, applied to a linear continuous problem, converges to a solution of dual multipliers at which the feasibility violations on dualized constraints are zero. At this point, the lower bound is equal to the upper bound (the feasible solution), therefore the original problem is solved optimally. This solution method, however, is commonly applied

Algorithm 1 Lagrangian relaxation algorithm

SLP2$(\mathbf{X}, \mathbf{Y}) \rightarrow$ SLP2$(\mathbf{X}, \mathbf{Y}, \lambda)$ {Relax and dualize (9), (10)}
$k \leftarrow 1$
repeat
  for all $s \in S$ **do**
    Solve SLP2$_s(\mathbf{X}, \mathbf{Y}, \lambda^k)$
  $Infeasibility \leftarrow (\mathbf{X}^k_s - \mathbf{X}^k_{s'}) + (\mathbf{Y}^k_{ts} - \mathbf{Y}^k_{ts'})$
  $k \leftarrow k + 1$
  $\lambda_k = \lambda^{k-1} + \delta^k$ {Update duals, continue dual search}
until $Infeasibility < tolerance$

to solve integer problems. In those cases, it works as a heuristic (Takriti and Birge (2000)).

2.2 L-shaped decomposition solution

The L-shaped decomposition algorithm, developed by Slyke and Wets (1969), has been extensively applied to both two stage and multistage SLP models. The multistage version is significantly more complicated than the two stage version. Thus, it deserves a complete study of its own, as demonstrated by Gassmann (1990), and Birge (1985). Consequently, we consider applications to two stage SLPs (see Bienstock and Shapiro (1988) for a comprehensive application example).

Consider a two stage version of the scenario tree in Figure 1: formulation SLP1 can be arranged into SLP3, with stage one and recourse parts explicitly identified and recognized. The L-shaped decomposition partitions SLP3 into a master problem and a subproblem as shown in the following formulations. The master problem includes the stage one problem and an additional variable θ which represents an approximation for the recourse problem. The subproblem is made up by the recourse problem. Notice that with this separation, the subproblem decomposes into independent scenario problems. The algorithm, outlined below, solves the master problem and the subproblems, iteratively. At each iteration, the solution to the subproblems is used to add a linear constraint in an effort to improve the approximation of θ for the recourse problem. These constraints, which are referred to as

Algorithm 2 L-shaped decomposition algorithm

$k \leftarrow 1$
repeat
  Solve Master$\Rightarrow \mathbf{X}^*, \theta^*$
  $(\mathbf{X}^k, \theta^k) \Leftarrow \mathbf{X}^*, \theta^*$
  for all $s \in S$ **do**
    Solve $\text{Sub}_s(\mathbf{X}^k)$
  Master += $\theta \geq g^k X + h^k$ {From the dual objective function}
  $k \leftarrow k+1$
until $\sum_{s \in S} z^*_{Sub_s(X^k)} - \theta^k \leq tolerance$

optimality cuts, are formed by the dual objective function of the subproblems at the current solution.

SLP3

$$
\begin{aligned}
& min \quad \mathbf{c}^x \mathbf{X} + min\ Q(\mathbf{X}) \\
& subject\ to \\
& \qquad \mathbf{AX} = \mathbf{b} \\
Q(\mathbf{X}) = min \quad & \sum_{s \in S} p_s \mathbf{c}_s^y \mathbf{Y}_s \\
& \mathbf{A}_s \mathbf{Y}_s = \mathbf{b}_s - \mathbf{B}_s \mathbf{X} \qquad \forall s \in S
\end{aligned}
$$

Master

$$
\begin{aligned}
& min \quad \mathbf{c}^x \mathbf{X} + \theta \\
& subject\ to \\
& \qquad \mathbf{AX} = \mathbf{b} \\
& \qquad \theta \geq g^k X + h^k \qquad \forall k \in K
\end{aligned}
$$

Sub($\mathbf{X}$)

$$
\begin{aligned}
& min \quad \sum_{s \in S} p_s \mathbf{c}_s^y \mathbf{Y}_s \\
& subject\ to \\
& \qquad \mathbf{A}_s \mathbf{Y}_s = \mathbf{b}_s - \mathbf{B}_s \mathbf{X} \qquad \forall s \in S
\end{aligned}
$$

Note that the recourse problem may be infeasible with respect to some of the solutions generated by the master problem. In this case, feasibility cuts are generated to restrict the feasible space of the master problem. This mechanism is very similar to that of the optimality cuts, therefore we omit discussion of feasibility cuts for brevity of exposition. A more complete description of the L-shaped algorithm can be found in Birge and Louveaux (1997).

2.3 Modeling language requirements

In order to support the Lagrangian relaxation algorithm, the nonanticipativity (NA) constraints should be accessible by the modeler through an abstract interface. The interface should provide functions for (*i*) accessing individual NAs associated with any node and inspecting their contents, (*ii*) dualizing the NA constraints, that is relaxing selected NAs and adding the relaxation to the objective function with appropriate dual multipliers, (*ii*) accessing the variables related to the NAs in a given node. These features would provide an interface to implement variations of the Lagrangian algorithm, either as exact or heuristic methods.

On the other hand, to support the L-shaped algorithm for two stage models, the language should provide an interface for (*i*) accessing the dual objective function of a given primal model, (*ii*) representing a model as a function of a given set of decision variables. These requirements can be best satisfied by an object oriented approach. Such an approach would be centered around a model class and a constraint class that would encapsulate all the elements of a SLP model and NA constraints respectively. These classes would provide a high level interface for the required functionality.

We base the proposed design on the open source AMPS package APLEpy, being developed by Karabuk (2005). There are two reasons for this choice, (*i*) the APLEpy software is open source, therefore the design can be implemented and made available to the community, (*ii*) the APLEpy software is based on the Python programming language, which provides an object oriented development environment that suits particularly well for implementing the requirements of the proposed design.

In the next section we first describe the current state of the APLEpy package, then we present the proposed language extensions for APLEpy.

3 The foundation of the design

The APLEpy package is made up of Python classes to represent model data, decision variables, objective function and constraints. A mathematical programming model can be described in Python, using built in language constructs and data types, and the APLEpy classes. We refer to a model coded in Python/APLEpy as an APLEpy model for brevity. Currently, the APLEpy package supports linear models, and SLP models. It is available as a free download at `sourceforge.net`, which is a large repository of open source projects.

3.1 An example problem

We consider the fabric production planning problem in textile manufacturing industry, develop an algebraic formulation that describes the problem, and briefly outline how to code it in APLEpy. A fabric manufacturing plant receives yarn as raw material and weaves it into fabric on machines called looms. Final fabric products are referred to as styles. Typically, a loom is capable of producing any style that the plant processes. Once a loom is configured for a style, it can produce it without any change until it is re-configured for another style, which requires a costly changeover process. A loom could be kept idle and preserve its configuration or it could produce to inventory in order to avoid changeover costs later on. The fabric production planning problem captures the trade off between inventory and changeover costs over a planning horizon of several periods. An important aspect of this decision problem is that demand for styles is a random variable.

The first step in developing a SLP formulation is to determine the random parameters, and stages at which they unfold. In this problem demand is the main source of uncertainty. The loom configuration decisions are made initially for the whole planning period independent of any particular demand realization. Thus, loom configuration, and changeover decisions are stage one decisions. They are associated with the root node of a scenario tree. Demand for a particular period (of all styles) become known at the start of a period, after which

production decisions are made, and inventories are observed as recourse. Thus, time periods in the planning horizon correspond to recourse stages. For example, stages 2 through 5 in Figure 1 correspond to periods 1 through 4 in this problem. This representation scheme captures the fact that production and inventory decisions of a planning period are delayed until after uncertainty clears in that period and the effects of decisions in the preceding period is observed, while uncertainty still exists in the succeeding periods.

Let i ($i = 1..N$) be the style index, t ($t = 1..T$) be the period index, and s ($s = 1..S$) be the scenario index. For simplicity, we use the same notation for the name of an index set and its cardinality, (e.g. $t = 1..T$ or $t \in T$). The decision variables are; X_{it}: the number of looms configured for style i in t, Z_{it}: the number of looms that are reconfigured for style i in t, Y_{its}: the number of looms that operate and produce style i in t under s, and I_{its}: the amount of inventory of style i carried in t under s. A loom either produces through a period at a constant rate of r_i or it is kept idle. Therefore, all decision variables except I are integer. Let c^I, and c^Z denote the inventory and changeover costs respectively. We also define m as the total number of looms available and d_{its} demand for style i in t under s. For brevity I_{i0s} and X_{i0} represent initial inventory and initial loom configuration respectively. We can develop a SLP formulation by using either the node based representation or a scenario based representation with explicit NA constraints, both of which are supported in APLEpy. We choose to proceed with the latter approach and present a formulation below.

Model SPP:

$$minimize \sum_{i\in N}\sum_{t\in T} c^Z Z_{it} + \sum_{i\in N}\sum_{t\in T}\sum_{s\in S} p_s c^I I_{its} \tag{3}$$

subject to

$$\sum_{i\in N} X_{it} = m \qquad \forall t \in T \tag{4}$$

$$X_{it} - X_{it-1} - Z_{it} \leq 0 \qquad \forall i \in N, t \in T \tag{5}$$

$$Y_{its} \leq X_{it} \qquad \forall i \in N, t \in T, s \in S \tag{6}$$

$$Y_{its} r_i + I_{it-1s} = d_{its} + I_{its} \qquad \forall i \in N, t \in T, s \in S \tag{7}$$

$$Y_{its} = Y_{its'} \qquad \forall s, s' \in \{NA_{ts} | (t,s) \in E\} \tag{8}$$

$$I_{its} = I_{its'} \qquad \forall s, s' \in \{NA_{ts} | (t,s) \in E\} \tag{9}$$

$$\mathbf{X}, \mathbf{Y}, \mathbf{Z} \in \mathbb{Z}^+, \mathbf{I} \in \Re^+$$

The objective (3) is to minimize the total of changeover and expected inventory carrying costs. In each period, every loom has to have a configuration, which is captured by (4). Constraints (5) track the number of changeovers in each period, and they make up stage one constraints together with (4). We choose to represent stage one decisions $(\mathbf{X}, \mathbf{Z})$ as they are, rather than defining one set for each scenario and imposing related NA constraints explicitly. This is because this representation differentiates stage one variables in a more intuitive way. Constraints (6), and (7), limit production capacity to the number of configured looms, and impose material balance relations respectively. These constraints are defined for each scenario individually, and they make up the recourse constraints. The NA constraints on recourse variables $\mathbf{Y}$, and $\mathbf{I}$ are described in (8), (9) respectively. The notation NA_{ts} refers to the set of scenarios that share node (t,s), and E refers to the set of event nodes, excluding the root node.

We think that model SPP presents a more intuitive formulation, compared to a node based formulation in which NA constraints are not needed explicitly. In Model SPP, the modeler's emphasis is on complete scenarios that represent one full description of the problem with respect to a complete state of future. The disadvantages, however, are that NA constraints are tedious to create manually, and that they increase the size of the problem considerably. The

APLEpy package effectively addresses these issues. It manages the NA constraints transparently to the user, and maps the scenario based formulation to a node based formulation internally, thus sending the same formulation to the solver in either formulation.

Next we show the means APLEpy provides for supporting SLP formulations, and a part of APLEpy code that implements Model SPP.

3.2 The existing SLP support in APLEpy

The APLEpy package is organized into modules. The LP module provides the basic functionality for description of a linear model, and the SLP module provides extensions to support SLP model description. A detailed description of the APLEpy's design and use can be found in Karabuk (2005). In the heart of the SLP module lies the `scenario_tree` class, which reads a specially formatted data file that describes a scenario tree. It creates all the required data structures and makes them available for use with APLEpy classes. The scenario data file constitutes one line of description for each node in the tree in the format shown in the first row of Table 1 with an example shown in row 2. Every node has to be labeled uniquely within a stage. The label could be either a number or an informative string that describes the significance of the associated event node. Parent node field refers to the label of the parent node at the previous stage. Node probability is the conditional probability of the associated event. The rest of the data fields constitute values of a random vector ($n > 1$) or a scalar ($n = 1$) associated with the node.

Table 1: Data input format for scenario tree

Stage	Node label	Parent node	Node probability	Rate1	Rate2	...	Rate$n$
2	Low_demand	High_demand	0.25	-0.2	0.05	...	0.4

Table 2 below shows the properties/methods provided by the `scenario_tree` class.

Table 2: Methods provided by `scenario_tree` class.

Property/Method	Return value
`event_nodes`	a list that contains event nodes E: $\{(t,s)\|(t,s) \in E\}$
`cprob`	a dictionary of path probabilities for each node (t,s)
`sprob`	a list of scenario probabilities for each scenario s
`prev`	a dictionary with key, value pairs as (t,s):`prev`(t,s)
`scenarios`	a list of scenarios
`stages`	a list of stages
`rate[i]`, $i \in R$	the i^{th} dictionary of values associated with each node (t,s)

The SLP module also provides classes `SLP.data`, `SLP.var` and `SLP.constraint`, which are derived from their counterparts in the LP module. The objects from these classes are initialized with an object of type `SLP.scenarios`, and optionally with an object of type `SLP.stages`, both of which are created by an `SLP.scenario_tree` object.

Model SPP can be implemented with the constructs described above. Listing (1) below shows the part of the complete APLEpy code that highlights SLP features. In the listing, Python keywords are underlined, and APLEpy classes are differentiated by prefix LP and SLP.

Listing 1: APLEpy code excerpt (Model SPP)

```
1  from APLEpy import LP
2  from APLEpy import SLP

4  Tree = SLP.scenario_tree('tree.dat')
5  p = Tree.sprob
6  SCEN = Tree.scenarios
7  WKS = Tree.stages

9  rate1 = Tree.rates[0]
10 ch = SLP.data(WKS, SCEN)
11 ch(rate1)

13 ...

15 Y = SLP.var(STYLS, WKS, SCEN, type=LP_integer, UB = total_looms)
16 I = SLP.var(STYLS, WKS, SCEN)

18 ...
```

```
20 capacity = SLP.constraint(STYLS, WKS, SCEN)
21 demand = SLP.constraint(STYLS, WKS, SCEN)

23 ...

25 Total_Expected_Costs.Minimize =
26   sum(cz*Z[i,t] for i in STYLS for t in WKS) +
27   sum(p[s]*ci*I[i,t,s] for i in STYLS for t in WKS for s in SCEN)

29 ...

31 for (i,t,s) in ((i,t,s) for i in STYLS for t in WKS for s in SCEN):
32   capacity[i,t,s] = Y[i,t,s] <= X[i,t]

34 for (i,t,s) in ((i,t,s) for i in STYLS for t in WKS for s in SCEN):
35   if t==1:
36     demand[i,t,s] = Y[i,t,s]*r[i] + Io[i] ==
37                                     d[i,t]*(1+ch[t,s]) + I[i,t,s]
38   else:
39     demand[i,t,s] = Y[i,t,s]*r[i] + I[i, t-1, s] ==
40                                     d[i,t]*(1+ch[t,s]) + I[i,t,s]

42 LP.Solve()
```

The syntax of the code is self explanatory, therefore we will focus on describing the functionality. Lines 1-2 import the required APLEpy modules. An object of type `SLP.scenario_tree` is created from input file 'tree.dat' in line 4. The scenario probabilities, and `scenarios` and `stages` objects are created and assigned to local variables in lines 5-7 respectively. The first rate vector is captured (line 9) and used to initialize an object of type `SLP.data` (line 11). In this example, the `rates` values represent fractional changes applied to a base demand vector (which is input separately) at each event node (t,s). Another approach would be to input as many rates fields as the total number of styles in which case the rate of change with respect to the base demand would be different for all styles. The input scenario tree excludes the root node, therefore, the stages returned by the `Tree` object (line 7) correspond directly to the planning periods. This is because, in this particular formulation we choose to handle stage one decisions, which are associated with the root node, explicitly.

The recourse variables and the recourse constraints are declared with scenarios as part of their

index set in lines 15-16 and 20-21 respectively. The objective function is described in lines 25-27. The formulation is completed with the description of the recourse constraints in lines 31-40. A model is compiled from all the elements described in the (complete) code and sent to the solver in line 42. Notice that NA constraints are not specified explicitly. The APLEpy objects handle them implicitly by consulting the scenario tree specifications. Technically, the implementation is based on intercepting the stage and scenario indices before they are used, in any manner, by the respective `SLP` objects, and map them to the corresponding node index in the scenario tree. The result is that the formulation that is passed on to the solver is the same as the node based compact formulation of the same problem.

4 The proposed design

In the current implementation of APLEpy, the NA constraints are implicitly managed in a transparent manner to the modeler. We develop an interface for accessing and manipulating the NA constraints that are managed behind the scenes. To this end, we add methods and properties to the `LP.constraint` class, and we create a new class `SLP.NAconstraint` derived from `LP.constraint`. The proposed additions are listed in Tables 3 and 4 for the two classes respectively.

Table 3: Functionality provided by the `LP.constraint` class.

Property/Method	Return/affect
`dual`	return the current dual value accept an assignment (writable)
`relax`	relax the constraint(s) in the model
`dualize`	relax, multiply with its dual and add to the objective function of the model
`infeasibility`	return the amount of feasibility violation at current values of decision variables

The `dual` property returns the current dual value associated with the subject constraint. We

Table 4: Functionality provided by `SLP.NAconstraint` class.

Iterator	Return value
`NActrobject[`*nodeindex*`]`	a single NA constraint at event node *nodeindex*
`NActrobject[`*nodeindex*`].NAVar()`	a list of decision variables (objects of type `LP.var`) of the same type at event node *nodeindex*

propose that this value be assignable as well, so that when the dual problem is solved by a user defined algorithm, its values can be updated externally. The `relax` method relaxes the subject constraints in the associated model, whereas the `dualize` method multiplies the relaxed constraints with their associated dual multipliers and places them in the objective function. The `infeasibility` method measures the amount of infeasibility of the relaxed constraint(s), at the current solution, and returns the value. All the properties and methods we discuss here apply to both a collection of objects or to a single object depending on the context of use. For example, the dualize method, if called by an object of type `LP.constraint` without any index, dualizes the whole collection of the constraints contained by the object. On the other hand, if it is called with a specific index, then it dualizes only one constraint referred to by the index.

The `SLP.NAconstraint` class provides additional functionality based on the special structure of the NA constraints. An object of type `SLP.NAconstraint` contains the whole collection of NA constraints. It cannot be created directly, but it is returned by an object of type `SLP.model`, which we will describe in detail below. It is indexed by event nodes only. As it is clear from the formulations of previous sections, each node index refers to a collection of NA constraints. Therefore, it is not possible to access a single NA constraint with a specific index. We picked this design, because (*i*) we were not able discover a meaningful abstraction to refer to individual NA constraints, and (*ii*) partly because of (*i*), we think that a modeler would not have a need to do so. However, we provide a general way of accessing and examining individual NA constraints by Python iterators. A Python object, of a class that defines an iterator, can be used in a loop returning elements from the collection of objects

that the iterator is defined on. The design provides two iterators with the `SLP.NAconstraint` class. One returns individual NA constraints from a given node index, and the other returns a collection of decision variables referred to by the same object.

For example, consider the scenario tree in Figure 1 and model SPP. Assume that the following are the complete set of NA constraints at node (3,4): $Y_{34} = Y_{35}$, $Y_{35} = Y_{36}$, $I_{34} = I_{35}$,$I_{35} = I_{36}$. The first iterator would return each of the four constraints sequentially. The second iterator would loop twice and it would return (Y_{34}, Y_{35}, Y_{36}), and (I_{34}, I_{35}, I_{36}) in sequence.

Similarly, we propose to add properties/methods to the `LP.model` class, and create a new class, `SLP.model`, derived from the `LP.model` class. Tables 5 and 6 list the proposed functionality for the two classes respectively.

Table 5: Functionality provided by `LP.model` class.

Property/Method	Return/affect
`objectiveFnValue`	the objective function evaluated at current solution
`infeasibility`	total amount of feasibility violation over all relaxed constraints at current solution
`dualObjectiveFn`	returns the objective function of the dual model as a function of the primal decision variables
call( )	
`LPmodelobject(`*`LPvar1 , LPvar2 ,.`*`)`	object of type `LP.model` as a function of argument
	variables, i.e. variables are fixed at current value

The first property we want to include in the `LP.model` class is `objectiveFnValue`. This is more of a convenience rather than a necessity. When dealing with more than one model (hence multiple objective functions), remembering the name of the objective function may be a chore, especially if a long descriptive name is given. This property of the class returns the value of the objective function at the current solution. The same value can be obtained by calling the objective function directly with its name. The next property, `infeasibility`, returns the total of infeasibility over all relaxed constraints, if any. This is a shortcut over

Table 6: Functionality provided by `SLP.model` class.

Property/Method	Return value
`nonAnticipativity`	object of type `SLP.NAconstraint`
`deterministicModel`	object of type `LP.model` that describes
	the expected value problem
`stageOne`	object of type `LP.model` for the stage one problem
`recourse`	object of type `LP.model` for the recourse problem
iterator	
`SLPmodelobject`	object of type `LP.model` for a scenario problem

calling all the relaxed constraints individually and summing their infeasibility results. The `dualObjectiveFn` property returns the objective function of the dual problem as a linear expression. It preserves the variables defined by the primal model and treats the dual variables as parameters evaluated at the optimal solution. This feature can be used to generate cuts in the L-shaped decomposition algorithm. The Python language provides overloading the function call operator, which is applied by matching parenthesis to an object. This feature enables an object to simulate a function call. We take advantage of this language feature to treat a model as a function of a set of variables. The call operator has the effect of fixing the values of the passed variables at their current solution. As a result they are treated as parameters during the solution of the model.

The proposed features of the `SLP.model` class provides a high level interface for extracting sub models from a given SLP model. The names of the properties are self explanatory. The deterministic model is formed by taking expected value of the random parameters and dropping the scenario index of recourse variables. The stage one and recourse models are formed in a straightforward fashion. The stage one problem collects variables with no scenario index, and constraints that do not include any recourse variables. The remaining part of the model constitute the recourse model. As such, this feature is most useful for two stage problems, or multistage problems that can be described as two stage models (Louveaux (1986)). However, this provides a foundation for extending the properties to multistage models. The

iterator defined on an object of type `SLP.model` returns individual scenario models as objects of type `LP.model`. Finally, the `nonAnticipativity` property returns an object of type `SLP.NAconstraints` that encapsulates the NA constraints of the SP model. Note that the `SLP.model` class is derived from `LP.model` class and it inherits all the methods/properties of its parent class.

Next, we illustrate the use of this design in implementing the decomposition algorithms we outlined in section 2.

4.1 The Lagrangian relaxation algorithm

Listing 2 below illustrates a basic implementation of the Lagrangian relaxation (LR) algorithm to solve the example problem of section 3.

Listing 2: Implementation of Lagrangian Relaxation Algorithm

```
1  SPP = SLP.model(Total_Expected_Costs,
2                  X, Z, Y, I,
3                  availability, changeover, capacity, demand)

5  AllNAs = SPP.nonAnticipativity
6  EVNODES = Tree.event_nodes

8  for node in EVNODES:
9    for NActrs in AllNAs[node]:
10      NActrs.dualize()

12 while True:
13   for sprob in SPP:
14     LP.Solve(sprob)

16   if SPP.infeasibility <= tolerance:
17     break

19   for node in EVNODES:
20     for NActrs in AllNAs[node]:
21       for ctr in NActrs:
22         ctr.dual = ctr.dual + ctr.infeasibility * delta
```

An object of type `SLP.model` is created, in lines 1-3, by passing the references to an objective

function (line 1), decision variables (line 2), and constraints (line 3). The NA constraints, and the set of event nodes needed to index them are extracted in lines 5, and 6 respectively. Lines 8-10 demonstrate how to access the NA constraints and how to dualize them. The `NActrs` reference in line 10 refers to a collection of NA constraints associated with the corresponding node index. Therefore, a set of NA constraints are dualized at the same time with one statement.

Line 12 starts the iterations for the LR algorithm. The `for` statement in line 13 invokes the iterator defined on class `SLP.model` and receives one model object each time it executes. The returned scenario model is then solved in line 14. A simple termination condition is checked in 16-17. The code in lines 19-22 accesses all the NA constraints individually and updates the duals, by making use of the infeasibility amount at each NA constraint. Line 21 demonstrates the use of iterating through individual NA constraints without specifying indices. The Python variables `tolerance` and `delta` are assumed to be user defined scalar parameters.

We provide, in listing 2, an outline which does not directly correspond to any specific LR algorithm application. A subgradient optimization based application would compute an upper bound and update the dual multipliers with a more involved mathematical formula. On the other hand, an augmented Lagrangian based application would look very similar to the listing, except that the dualized terms in the objective function would be quadratic. The means to complete both algorithms already exist in Python/APLEpy. We leave some of the details, such as providing an easier way of specifying the form of dual terms in the objective function, for implementation.

4.2 The L-shaped decomposition algorithm

Listing 3 below shows implementation of the L-shaped decomposition algorithm for a two stage SLP.

Listing 3: Implementation of L-shaped Decomposition Algorithm

```
1  Master = SPP.stageOne
2  Sub = SPP.recourse

4  While True:
5    LP.Solve(Master)

7    recoursedualObjectiveFn = 0
8    for sprob in Sub(X, Z):
9      LP.Solve(sprob)
10     recoursedualObjectiveFn += sprob.dualObjectiveFn

12   if (Sub.objectiveFnValue - theta) <= tolerance:
13     break
14   else:
15     Master += theta >= recoursedualObjectiveFn
```

The master problem (stage one) and the subproblem (recourse) are extracted from the original model referred to by the SPP object in lines 1-2. Line 4 starts the iterations of the decomposition algorithm. The master problem is solved in line 5, and the individual scenario problems that make up the subproblem are solved one at a time through lines 8-10. At the same time, the dual objective function of the subproblem is constructed progressively from the solution of scenario models. Notice that the subproblem is called with stage one decision variables as parameters (line 8). In line 12, the gap between the upper bound (objective function value of the subproblem) and the lower bound (theta value) is checked against a predefined tolerance value. If the termination criterion is not satisfied, then, in line 15, an optimality cut is added as a linear constraint to the master problem. We assume that an additional variable, theta, has been defined and added to the objective function of model SPP which is then passed to the master problem. If applied to a SLP that does not have full recourse (see Birge and Louveaux (1997)), the algorithm would include an additional step for generating feasibility cuts, because some of the stage one solutions would cause an infeasible recourse problem. We exclude this part for brevity, however, the means to complete the

infeasibility step is in place.

The stochastic production planning model we defined in the previous section could also be solved by the L-shaped decomposition algorithm, because it has block separable property. However, the recourse part is still multistage and therefore does not decompose into independent scenario problems. Hence, the `for` loop in lines 8-10 would reduce to a single statement solving the subproblem as a whole.

4.3 Developing heuristic algorithms

Although we specifically focus on supporting mathematical decomposition algorithms for solving SLP models, the resulting design is general in that it can be used to implement other heuristic algorithms. In order to illustrate this, we first show in listing 4 below, the calculation of EVPI (expected value of perfect information) and VSS (value of stochastic solution) values for SLPs (see Birge and Louveaux (1997)). The EVPI and VSS are standard measures that are used to quantify the benefits and the quality of applying a SLP model respectively.

Listing 4: Computation of VSS and EVPI values

```
1  LP.Solve(SPP)
2  SP = SPP.objectiveFnValue
3  PI = 0
4  for sprob in SPP:
5    LP.Solve(sprob)
6    PI += sprob.objectiveFnValue
7  EVPI = SP - PI

9  DPP = SPP.deterministicModel
10 LP.Solve(DPP)
11 LP.Solve(SPP(X, Z))
12 EEV = SPP.objectiveFnValue
13 VSS = EEV - SP
```

Lines 1-7 compute the EVPI value, which is the solution to the original SLP model (SPP) less the PI (perfect information) value. The PI value is computed by solving each scenario problem independently and summing their objective function values (lines 4-6). Notice that

there is no need to relax NA constraints to iterate over SPP. The `SLPmodel` object handles this internally. The VSS value, which is the EEV (expected value of expected value solution) less the solution value to SPP, is computed in lines 9-13. The EEV value is computed in two steps: (*i*) extract the deterministic correspondent to SPP and solve it (lines 9-10), (*ii*) solve model SPP as a function of the stage one solution computed by the deterministic model (DPP).

Next, in listing 5, we illustrate an implementation of a simple heuristic, which is inspired by the rounding heuristic commonly applied to solve integer models. The original heuristic relaxes integrality requirements, and iteratively solves and rounds of fractional values until a complete integer-feasible solution is achieved.

Listing 5: A Heuristic Rounding Algorithm for SLP

```
1  EVNODES = Tree.event_nodes
2  Varlist = []

4  AllNAs = SPP.nonAnticipitivity
5  AllNAs.relax()

7  while True:
8    for sprob in SPP(Varlist):
9      LP.Solve(sprob)

11   if SPP.infeasibility <= tolerance:
12     break

14   for node in EVNODES:
15     for NActrs in AllNAs[node]:
16       if (NActrs.infeasibility < limit):

18         for NAvars in AllNAs[node].NAvar():
19           average = sum(NAvars) / len(NAvars)
20           for var in NAvars:
21             var = average
22             Varlist += var
```

The set of event nodes are extracted for use in indexing the NA constraints, and an empty list is created in lines 1, and 2 respectively. The purpose of the object `Varlist` is to contain references to the decision variables whose values are fixed by the algorithm. Next, NA

constraints are extracted and relaxed (lines 4-5). The iterations start in line 7. The first step is to solve all the scenario problems (lines 8-9), and check overall infeasibility in SPP (line 11). Notice that SPP object is called with a list of decision variables whose values are fixed. Therefore, the scenario models returned by the iterator defined on SPP assume the fixed variables as parameters. Through lines 14-16, the total of infeasibility over the collection of NA constraints at each event node is checked and compared against a predefined `limit` value. If the overall infeasibility at a node is under the limit value, then, through lines 18-22, the values of variables involved with the NA constraints at the node, are averaged and fixed. The `for` statement in line 18, demonstrates the use of the second iterator (row 2 of Table 5) defined on object of type `SLP.NAconstraint`. At each iteration of the `for` loop, `NAvars` refers to a list of variables of the same name, but of different indices. The Python built in functions `sum()` and `len()` return the sum of the current values of the variables in the list, and the total number of variables in the list respectively. The `for` loop in line 20 iterates over individual decision variables in the list, assigns the average to each, and adds to the list of fixed variables. The `while` loop continues until the termination criterion in line 11 in satisfied.

This solution algorithm, in its general form, would not be applicable to most SLP problems. This is because, as the decision variables are progressively fixed, the remaining problem could be infeasible. However, as with the original heuristic that inspired this one, this is simply a framework which should be applied with problem specific details. Nevertheless, it clearly demonstrates the use of the language constructs we have designed for creating and implementing algorithms for solving SLP models.

As another example, consider a nested decomposition approach for solving model SPP of the previous section. We apply L-shaped decomposition as the first level, and apply Lagrangian relaxation to solve the subproblem as the second level. This can easily be implemented based on the code in listings 2 and 3.

An important consequence of the object oriented nature of the design and of the APLEpy

environment is that, the code for an algorithm can be used with little or no modification to solve another model that shares the same assumptions. This feature can be clearly observed from the code listings. There are no problem specific references to any model elements, with the exception of specifying the names of stage one variables in listing 3.

5 Conclusion

We present a design for extending algebraic modeling languages in support of developing solution algorithms at a high abstraction level. The design is object oriented and centered around a model class for a stochastic linear program and a constraint class for nonanticipativity conditions. These classes provide an abstract interface in manipulating their contents for well defined tasks that would otherwise be done by a modeler. For example, consider the `dualObjectiveFn` method of `(S)LP.model` class. A modeler could go through the whole formulation and construct the dual objective function manually by applying well defined rules. However, these kinds of well defined and repetitive tasks are best handled by the development environment. Therefore, the contribution of this design, if implemented, is to considerably reduce algorithm development time for solving large scale SLP models.

A byproduct of this design is to introduce the concept of a special purpose constraint type, namely for nonanticipativity constraints. There are specially structured constraints in many model types, such as inventory balance constraints in production planing problems, or material flow balance constraints for network models. In current AMLs all the problem constraints are described explicitly by a modeler. The next level of abstraction would be to specify the type of a constraint, which would then be created and maintained by the development environment. An interface would also be provided for querying and manipulating its contents by the modeler. We can extend this concept further to the model level by providing means to specify a model type (e.g. transportation, or multi-stage SLP etc). The environment would then create a model object that includes base features that are common to all instances of

the specified type. The modeler's task would then be reduced to specifying the input data and the non-standard features of the particular model.

The immediate next step in this research is to implement the design in APLEpy environment. The design is created with implementation in mind, therefore there are neither technological nor licensing limitations for implementation. The implementation should provide more insights, with which the main concept can be extended beyond SLP models. In fact, the design could already handle general models, because the main functionality is encapsulated in the core LP classes. However, it requires a through testing before we can make this conclusion. Of particular importance for future research is to cover nested application of the L-shaped algorithm to solve multistage problems. As it is, the complexity of this algorithm prevents it from becoming widely practised.

According to this author, a major use of this design is for teaching. One of the better ways of teaching SLP and associated solution algorithms is to demonstrate their use with software. In this author's experience it takes a long time for students to be comfortable with the NA constraints and be able to implement decomposition algorithms in any AML. The APLEpy environment, with the proposed design, would reduce the learning time in such courses.

References

Beasley, J. E. (1993). Lagrangean relaxation, *in* C.R.Reeves (ed.), *Modern heuristic techniques for combinatorial problems*, Blackwell Scientific Publications, chapter 6, pp. 243–303.

Bienstock, D. and Shapiro, J. F. (1988). Optimizing resource acquisition decisions by stochastic programming, *Management Science* **34**(2): 215–229.

Birge, J. and Louveaux, F. (1997). *Introduction to stochastic programming*, Springer-Verlag.

Birge, J. R. (1985). Decomposition and partitioning methods for multistage stochastic linear programs, *Operations Research* **33**(5): 989–1007.

Brooke, A., Kendrick, D., Meeraus, A. and Raman, R. (1998). *GAMS: A user's guide*, GAMS Development Corporation.
URL: *http://www.gams.com*

Buchanan, C., McKinnon, K. and Skondras, G. (2001). The recursive definition of stochastic linear programming problems within an algebraic modeling language, *Annals of Operations Research* **104**: 15 – 32.

Domenica, N. D., Birbilis, G., Mitra, G. and Valente, P. (2004). Stochastic programming and scenario generation within a simulation framework: An information system perspective, *The Stochastic Programming E-Print Series (SPEPS)* (15).
URL: *http://hera.rz.hu-berlin.de/speps*

Dominguez-Ballesteros, B., Mitra, G., Lucas, C. and Koutsoukis, N.-S. (2002). Modelling and solving environments for mathematical programming (mp): A status review and new directions, *The Journal of the Operational Research Society* **53**(10): 1072–1092.

Entriken, R. (2001). Language constructs for modeling stochastic linear programs, *Annals of Operations Research* **104**: 49 – 66.

Fourer, R. (1996). Proposed new ampl features: Stochastic programming extensions, World Wide Web.
URL: *http://www.ampl.com/NEW/FUTURE/stoch.html*

Fourer, R. (2005). Linear programming: Eighth in a series of lp surveys highlights recent trends in profession's most popular software, OR/MS Today.
URL: *http://www.lionhrtpub.com/orms/orms-6-05/frsurvey.html*

Gassmann, H. I. (1990). Mslip: A computer code for the multistage stochastic linear programming problem, *Mathematical Programming* **47**: 407–423.

Gassmann, H. I. (1998). Modeling support for stochastic programs, *Annals of Operations Research* **82**: 107–137.

Gassmann, H. I. and Ireland, A. M. (1995). Scenario formulation in an algebraic modeling language, *Annals of Operations Research* **59**: 45–75.

Gassmann, H. I. and Ireland, A. M. (1996). On the formulation of stochastic linear programs using algebraic modeling language, *Annals of Operations Research* **64**: 83–112.

Guéret, C., Prins, C. and Sevaux, M. (2002). *Applications of optimization with Xpress-MP*, Dash Optimization.

Kallrath, J. (ed.) (2004). *Modeling Languages in Mathematical Optimization*, Vol. 88 of *Applied Optimization*, 1st edn, Springer.

Karabuk, S. (2005). An open source algebraic modeling and programming software, *Technical report*, University of Oklahoma, School of Industrial Engineering.

Karabuk, S. and Wu, S. D. (2002). Decentralizing semiconductor capacity planning via internal market coordination, *IIE Transactions* **34**(9): 743, 17 pgs.

Kristjansson, B. (2003). Using the optimax 2000 component library to create integrated end-user applications, INFORMS National Meeting. Atlanta, Georgia.
URL: *http://www.maximal-usa.com*

Louveaux, F. V. (1986). Multistage stochastic programs with block-seperable recourse, *Mathematical Programming Study* **28**: 48–62.

Rockafellar, R. T. and Wets, R. J.-B. (1991). Scenarios and policy aggregation in optimization under uncertainty, *Mathematics of Operations Research* **16**(1): 119–147.

Slyke, R. V. and Wets, R.-B. (1969). L-shaped linear programs with application to optimal control and stochastic programming, *SIAM Journal on Applied Mathematics* **17**: 638–663.

Takriti, S. and Birge, J. R. (2000). Lagrangian solution techniques and bounds for loosely coupled mixed-integer stochastic programs, *Operations Research* **48**(1): 91–98.

Valente, P., Mitra, G., Poojari, C. A. and Kyriakis, T. (2001). Software tools for stochastic programming: A stochastic programming integrated environment (spine), *Tr/10/01*, Department of Mathematical Sciences, Brunel University, West London, UK.

Valente, P., Mitra, G. and Sadki, M. (2004). *SAMPL/SPInE: User Manual*, OptiRisk Systems, Middlesex, UK.

Valente, P., Mitra, G., Sadki, M. and Fourer, R. (2005). Extending algebraic modelling languages for stochastic programming, *The Stochastic Programming E-Print Series (SPEPS)* (8).

URL: *http://hera.rz.hu-berlin.de/speps*