AML: Algebraic Markup Language

Obi C. Ezechukwu
Department of Computing
Imperial College London
Exhibition Road
London SW7 2BZ
United Kingdom

mailto:oce@doc.ic.ac.uk

Istvan Maros
Department of Computing
Imperial College London
Exhibition Road
London SW7 2BZ
United Kingdom

mailto:im@doc.ic.ac.uk

Departmental Technical Report 2003/12

ISSN 1469-4174

December 2003

Abstract

There exists a variety of formats for representing optimisation models, each with varying degrees of support on optimisation platforms. Existing formats are also unsuitable for Internet distributed computing technologies, due not only to their syntax but also to limited degrees of portability. This paper describes an XML based algebraic mark-up language for representing optimisation models. The principal aim of the language is to provide an abstracted representation format with sufficient generality to capture the structure of optimisation models, and which can easily be ported to existing formats, and is usable in an Internet computing environment.

Keywords: *XML, XSD, Optimization, Modelling, Web Services*

Since the mid 1990s OR practitioners have been looking to harness the strengths of the Internet for the benefit of the OR community. As such, a lot of effort [Bhargava and Krishnan (1998), Fourer and Goux (2001), Kristjansson (2001)] has been directed at exploiting technologies such as web services in providing optimization services. Existing model representation formats are incompatible with recent Internet technologies and standards e.g. SOAP [17] and XML [21]. Consequently, they cannot be readily used in a web services or Internet distributed computing environment.

Optimization models and model instances can be encoded in a variety of ways, including algebraic notation in an electronic document e.g. word document or rtf; a modelling language supported by some modelling system such as GAMS [20], AMPL [Fourer, Gay, and Kernighan (1993)], or MPL [13]; high level programming language structures; or a low level input format i.e. matrix level format. Low-level matrix formats are typically used for representing *model instances* as opposed to models, where the term *model instance* refers to a model where values have been assigned to the data structures that constitute the model. Clearly, a model instance offers less reuse than the model itself, which can be repeatedly solved with different data sets. Hence, most modelling systems aim for a separation of the model and the data required to create an instance of the model.

In general, the encoding schemes used for representing optimization models are too closely aligned with the ultimate target or use of the model. Computational formats are not portable and closely couple model implementations to specific platforms or systems, consequently reducing portability and reuse of models. This makes it very difficult to use such representation formats in Internet based distributing computing solutions/applications. In the case of algebraic formats i.e. modelling languages, although these are text based and text can be sent over Internet protocols, in a SOAP environment for example, it would be difficult to validate what it is that is being sent. There would be no structured way of telling the difference between the string *'Hello World'* and an algebraic model. Achieving this would involve implementing some sort of parser at the receiving end of the transport, which would be no mean exercise as it would essentially involve replicating the functionality of the target modelling system. Whereas in a profoundly extreme case a weak argument may be made for doing this for inter-organisation applications, in a wider e-commerce context the risks and costs of such a scheme would very quickly overwhelm any argument for it.

A related argument can be made against callable libraries. Although distributed software technologies exist for different platforms, such as Microsoft's DCOM for windows platforms, which can be used to provide distributed access to optimization components, they are not particularly portable and do not fit easily into the web services architecture. Ultimately these can and generally should be used to provide the backbone for web services, however a representation format is still required to transport models and model data across the Internet. One of the aims of web services is to allow the interaction of heterogeneous computer systems, regardless of hardware or operating system platform, or programming language. The use of hardware, language or operating system specific toolkits or technologies defeats this purpose.

As such, a text based representation scheme is required for representing models that fits in seamlessly with existing Internet technologies, standards and architectures. However rather than building the functionality for a representation format from scratch, it is desirable to re-use the wealth of functionality that has already been implemented for existing formats. Moreover given the number of available modelling systems, which already do a good enough job representing optimization models and model data, it would be quite difficult to justify the addition of another modelling system. The aim is not to create a modelling system or callable library but rather a representation format, which is either an effective abstraction of, or encapsulates the functionality provided by most modelling languages. Our preferred approach is to create an abstraction of existing formats using a meta-language such as XML. Such a format would enable the generation of multiple views from a single representation, where a view could be algebraic modelling language code, instance of high level programming language structures, or even the algebraic form of the model i.e. the conceptual model, thus allowing the model to be ported to specific systems or platforms at execution time. *Ignoring the intricacies of optimization models, it is possible to define an encoding scheme which abstracts away*

from the current forms of representation, and which can be used to generate multiple views. In essence, capture the information about the model without tying the representation either to an execution environment or to a particular view or use. It is highly unlikely that all the product vendors in the market today can be encouraged to adopt a single format for representation, so it is important that the format doesn't explicitly require vendor support.

AML achieves this by building on the earlier work on *SML (Structured Modelling Language)* [Geoffrion, 1987] and leveraging the strengths of XML to provide an algebraic mark-up language that not only abstracts away from existing representation formats, but which can easily be ported to existing algebraic modelling languages. AML is part of a wider suite of technologies collectively referred to as the 'Open Optimization Framework' [Ezechukwu and Maros, 2003], which provide an integrated solution for model representation, distributed optimization over web services, and the integration of optimization software. The framework is intended as an open source project and as such, is geared heavily in the direction of open standards such as XML and platform independent programming languages such as Java.

The following sections provide an overview of the AML objectives, an example model, and a description of the AML syntax in the context of the given example. It also covers briefly some of the implementation decisions that have been made with regards to the language, in particular validation, and support for vendor specific extensions. The language grammar is encapsulated in a set of XML schemas available from [1]. XSL stylesheets have also been implemented for a number of algebraic modelling systems, namely GAMS, AMPL, and MPL, in order to demonstrate the portability of AML, and these are also available from [1]; given an AML model and its instance data, these stylesheets will transform it to the respective algebraic modelling language format.

1 Objectives

The main aim of AML is to provide an abstraction of existing model representation schemes, which can be used in conjunction with Internet architectures such as web services. Put in simple terms, in addition to representing optimization models and model data, AML aims to provide a typing mechanism, which essentially restricts what does and doesn't qualify as an optimization model.

The aims driving the development of the language can be summarised in the following points:

i. **Abstraction of existing formats:** AML is not a modelling language nor is it part of any modelling system; rather it is an abstraction of the structure of an optimization model. AML aims to provide a portable representation format, which enables the portability of models across modelling systems, platforms and programming languages. XML provides a means of achieving this as its syntax primarily enforces the representation of data using a tree structure. This can be extended in order to achieve the graph structure of optimization models advocated in SML, and therefore provide an easy means of replicating the algebraic structure of optimization models. Moreover, existing algebraic modelling systems share a number of similarities, which means that it is possible to extract a common syntax structure from them. The abundance of freely available XML processing tools makes it relatively easy to map the representation to callable library routines or structures if desired, and also provide developers runtime access to the model and model data at relatively little cost. The drawback of the abstraction philosophy however, is that individual strengths of particular systems could be lost in the process.
ii. **Separation of presentation and processing:** AML does not process models or instance data; rather it simply provides a means of representing models. In order to process the model it has to be transformed to a format, which is accepted by a target optimisation system. To ensure portability, AML is not tied to any particular representation format or modelling system, rather the language is organised in such a way as to enable the easy transformation of AML models and model data to algebraic modelling language formats, or high level programming language structures. This can be accomplished in a number of ways depending on developer preference including the use of programming constructs, however XSL is designed primarily for this purpose i.e. transforming

XML content, and quite possibly provides the most convenient means of achieving the desired goal.

iii. **Compact syntax:** The AML syntax is designed to be compact and easy to comprehend. It is also designed to closely mirror not just the algebraic structure of models and their associated data, but also the structure of existing algebraic modelling systems. AML is self-contained in the sense that it doesn't reference or rely upon structures outside the core XML defined data types. As such, it does not make use of MathML or any other such mathematical mark-up language. The reason for this is twofold, the first of which is to limit the scope of the applicable syntax as already mentioned, but the latter is related to validation and typing, which will be explained in the following section.

iv. **Grammar defined validation:** In addition to the structure of the language, the AML grammar attempts to encapsulate as much of the validation required for models as possible. This is to reduce the need for additional validation routines at the software layer. In order to achieve this, a decision was taken to avoid using much more loosely typed constructs from other mark-up languages such as MathML. Whereas MathML components would serve well in representing mathematical notation, they would be a little ambiguous in the context of an application area as specific as optimization and increase the validation required at the software level.

v. **Extensibility:** By building on XML, AML allows easy extension of its syntax. This is particularly useful if a user/organisation wants to extend AML to map directly onto a specific modelling system, e.g. by extending the list of inbuilt functions. Hence, although AML is intended as an abstraction of existing representation formats, there's nothing that stops the users from extending its syntax to cater for a specific modelling or programming language. This is further encouraged by the fact that its constituent schemas and associated software are distributed in source format.

2 Example Model

This section presents the multi-period production-planning model, which provides the context for the syntax description given in the following section. The objective of the model is to maximise the net revenue or profit from the production of a number of items over a time period, subject to constraints on inventory and production capacity.

This example is provided in order to illustrate the full structure of an AML model, i.e. to provide basic examples of all the types of elements that may occur in an AML model. The conceptual model is provided in figure 2.1. The formulation contains three decision variables which can be explained as follows:

- *Store:* This is the amount of each product that is left in inventory at the end of each production period i.e. unsold stock
- *Sell:* The quantity of each product that is sold in each period
- *Produce:* The quantity of each product produced in a given time period

The model parameters i.e. data elements are as follows:

- *Price:* Sale price of each product
- *Demand:* Periodic product demand
- *ProductionCost:* The production cost per product
- *ProductionRate:* Production rate per product
- *WorkingDays:* The number of working days, or productive days in each period
- *StorageCost:* Per product storage cost
- *StorageCapacity:* Fixed storage capacity

It is obvious from the above statements that there are two sets namely:

- *Product:* The set of products
- *Period:* The set of time periods

$$\textbf{MAX } GrossRevenues - TotalCosts \quad (1)$$

$$\textbf{Subject to: } \sum_i (Produce_{i,j} / ProductionRate_i) \le WorkingDays_j \quad (2)$$

$$\sum_{j,i} Store_{i,j} \le StorageCapacity \quad (3)$$

$$Produce_{i,j} + Store_{i,j-1} = Sell_{i,j} + Store_{i,j} \quad (4)$$

$$Sell_{i,j} \le Demand_{i,j} \quad (5)$$

$$i \in products, j \in period \quad (6)$$

$$GrossRevenues := \sum_{i,j} Price_i Sell_{i,j} \quad (7)$$

$$TotalProductionCosts := \sum_{i,j} ProductionCost_i Produce_{i,j} \quad (8)$$

$$TotalStorageCosts := \sum_{i,j} StorageCost_i Store_{i,j} \quad (9)$$

$$TotalCosts := TotalProductionCosts + TotalStorageCosts \quad (10)$$

Figure 2.1: Production Planning Model

3 Syntax Description

AML maintains a very clear distinction between an optimization model, and the data required to instantiate it. This enables the re-use of a single model representation with multiple data sets. An AML model and its instance data are specified in two separate XML documents.

The model document begins with the top-level element:

```
<aml:optimizationModel modelId="ProductionPlanning"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:aml="http://www.doc.ic.ac.uk/~oce/elsinore/2003/AML">
```

The model can consist of up to six top-level nodes, ordered as follows: **"<*sets*>"**, "**<*parameters*>"**, **"<*variables*>"**, **"<*macros*>"**, **"<*objective*>"**, and **"<*constraints*>"**. At its bare minimum an AML model does not have to contain any elements, however from a practical point of view an optimisation model must contain an **"<*objective*>"** node, or in the case of a constraint satisfaction problem, the model must contain a "**<*constraints*>"** element. All models must specify a value for the *'modelId'* attribute. This provides a means for target systems to uniquely identify the model. It also encourages documentation on the part of the user. Expressions in AML are encapsulated by the *"<function>"* element, which can occur in macro, constraint and objective declaration. It can also occur in parameter declarations with the restriction that the encapsulated expression cannot include any references to variables or macros.

The instance data document begins with the following element:

```
<aml:optimizationModelData modelId="ProductionPlanning" modelInstanceId="Illustration"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:aml="http://www.doc.ic.ac.uk/~oce/elsinore/2003/AML">
```

At the minimum an AML instance data document should provide data for all the elements in the model that require a data assignment. Else, the model will not be considered a valid instance, for the principal reason that it would not be possible to obtain a solution to it. There are only two element

types in an AML model to which the modeller can assign data values to i.e. *"<set>"* and *"<parameter>"*elements. Values for these are provided in the ***"<setData>"*** and ***"<parameterData>"*** section of the data document. The reader will notice that in addition to the *'modelId'* element, the data document can also include an additional *'modelInstanceId'* attribute. This is provided for cases where the model is solved repeatedly using different data sets, for example when evaluating different scenarios or while performing analysis.

3.1 Sets

Sets are declared within the *"<sets>"* model node. An individual set is declared with a *"<set>"* tag, which requires that a unique id *'setId'* and *'alias'* be provided for the set. The *'alias'* attribute is provided for compatibility with some modelling systems which require that an alias is used to reference a set as opposed to the set identifier or name. Once a set is declared it can be referenced in two contexts: indexing expressions; and subscript expressions. The former occurs in the declaration portion of elements e.g. indexed variables, parameters or constraints, or when representing a function which iterates over the set. A subscript expression occurs in parameter and variable references, i.e. within expression elements. In both cases a set is referenced using the *"<index>"* tag which takes a *'setId'* attribute which should be set to the same value as the same named attribute within the set declaration.

The set declarations in the example model are given by:

```
<sets>
   <set setId="clothing" alias="c"/>
   <set setId="accessories" alias="a"/>
   <set setId="product" alias="p">
      <setOperation   operationId="UNION"
                  leftSetId="clothing"
                  rightSetId="accessories"/>

   </set>
   <set setId="period" alias="q"/>
</sets>
```

The example model in reality consists of two sets only, namely the *product* and *period*, however in order to illustrate the use of set operations, we have split the product set into two sets, *clothing* and *accessories*, and defined *product* as a union of both. This is done using a *'<setOperation>'* element within the *"<set>"* element that defines the *product* set. The *'operationId'* can also accept the values *INTERSECTION*, and *DIFFERENCE*.

In the case of the sets not defined by set operations i.e. *clothing*, *accessories*, and *product*, it is necessary to specify the elements of the set in the instance data document. The corresponding data element declarations are given by:

```
<setData>
   <!--products-->
   <setContents setId="clothing">
      <subscript>Trousers</subscript>
      <subscript>Shirts</subscript>
   </setContents>
   <setContents setId="accessories">
      <subscript>Socks</subscript>
   </setContents>

   <!--period identifiers-->
   <setContents setId="period">
```

```
      <subscript>1</subscript>
      <subscript>2</subscript>
      <subscript>3</subscript>
      <subscript>4</subscript>
    </setContents>
  </setData>
```

The *"<setData>"* tag encapsulates the data element declarations for all the sets in a given model. For each set, the *"<setContents>"* tag encapsulates the set elements, where each element is defined by the use of the *"<subscript>"* tag. The *"<setContents>"* tag requires that the name of the declared set be specified as the value of the *'setId'* attribute. This enables the processing system to associate the data elements to a given set. In AML, the elements of a set are referred to as *subscripts*.

3.2 Parameters

Parameters are declared within the *"<parameters>"* model element. Each parameter is declared with a *"<parameter>"* element specifying a value for the *'parameterId'* attribute, which plays the role of a unique identifier or name. In the case of multi-dimensional i.e. indexed parameters, the *"<parameter>"* can also contain indexing expressions i.e. the *"<index>"* element, where each index refers to a set over which the parameter is indexed. The parameter declarations for the example model are given by the following sequence of nodes:

```
  <parameters>
    <parameter parameterId="Price">
      <index setId="product"/>
    </parameter>

    <parameter parameterId="Demand">
      <index setId="product"/>
      <index setId="period"/>
    </parameter>

    <parameter parameterId="ProductionCost">
      <index setId="product"/>
    </parameter>

    <parameter parameterId="ProductionRate">
      <index setId="product"/>
    </parameter>

    <parameter parameterId="WorkingDays">
      <index setId="period"/>
    </parameter>

    <parameter parameterId="StorageCost">
      <index setId="product"/>
    </parameter>

    <parameter parameterId="StorageCapacity">
      <function>
        <numericLiteral>800</numericLiteral>
      </function>
    </parameter>
  </parameters>
```

Although the value of the *StorageCapacity* parameter can be specified in the data document, it is represented here as the result of a very basic expression. This is to illustrate the fact that expressions can be provided for the calculation of parameter values. This can be done by use of the *"<function>"* tag within the *"<parameter>"* node.

In the context of a parameter, a function element encapsulates an expression that defines the value(s) of the parameter. Such an expression can include numeric literals, calls to pre-defined arithmetic functions e.g. *"LN"* or *"SIN"*—a full list of the available functions can be obtained by examining the AML schemas. It can also include indexed set operations—functions which cause expressions to be repeatedly evaluated for each subscript combination from a number of sets. It can also include references to other parameters, and subscript operations—functions which operate the subscripts of a set.

It is also possible to nest parameter *"<function>"* tags within each other to any degree thus enabling the construction of complex expressions. However in a case where AML is used primarily for the integration of software components, it may be advisable to perform all data manipulation within the source (originator) software, as the programming language in which it is written is more likely to offer better data access and management subroutines that can ever be matched in a modelling environment. General expressions are covered in more detail in section 3.7.

In the case where there is no expression provided for calculating the value(s) of a parameter, the parameter data must be specified within the *"<parameterData>"* element of the AML data document.

The corresponding data values for the declared parameters are presented in tables 3.1 to 3.6 below.

	Trousers	**Shirts**	**Socks**
Price	39.50	35.00	5.99

Table 3.1:'Price' parameter values

	Trousers	**Shirts**	**Socks**
Period 1	1300	1750	4000
Period 2	800	1300	3000
Period 3	6000	4000	9000
Period 4	3000	8000	15000

Table 3.2:'Demand' parameter values

	Trousers	**Shirts**	**Socks**
Production Cost	20.00	21.00	6.00

Table 3.3:'ProductionCost' parameter values

	Trousers	**Shirts**	**Socks**
Production Rate	10	9	30

Table 3.4:'ProductionRate' parameter values

	Period 1	**Period 2**	**Period 3**	**Period 4**
Working Days	20	22	20	19

Table 3.5:'WorkingDays' parameter values

	Trousers	**Shirts**	**Socks**
Storage Cost	0.50	0.50	0.01

Table 3.6:'StorageCost' parameter values

The equivalent AML listing is as follows:

```
<parameterData>
   <!--product prices-->
   <parameterValues parameterId="Price">
      <parameterValue value="39.50">
         <subscript>Trousers</subscript>
      </parameterValue>
      <parameterValue value="35.00">
         <subscript>Shirts</subscript>
      </parameterValue>
      <parameterValue value="5.99">
         <subscript>Socks</subscript>
      </parameterValue>
   </parameterValues>

   <!--Demand for each product-->
   <parameterValues parameterId="Demand">
      <parameterValue value="1300">
         <subscript>Trousers</subscript>
         <subscript>1</subscript>
      </parameterValue>
      <parameterValue value="800">
         <subscript>Trousers</subscript>
         <subscript>2</subscript>
      </parameterValue>
      <parameterValue value="6000">
         <subscript>Trousers</subscript>
         <subscript>3</subscript>
      </parameterValue>
      <parameterValue value="3000">
         <subscript>Trousers</subscript>
         <subscript>4</subscript>
      </parameterValue>

      <parameterValue value="1750">
         <subscript>Shirts</subscript>
         <subscript>1</subscript>
      </parameterValue>
      <parameterValue value="1300">
         <subscript>Shirts</subscript>
         <subscript>2</subscript>
      </parameterValue>
      <parameterValue value="4000">
         <subscript>Shirts</subscript>
         <subscript>3</subscript>
      </parameterValue>
      <parameterValue value="8000">
         <subscript>Shirts</subscript>
         <subscript>4</subscript>
      </parameterValue>

      <parameterValue value="4000">
         <subscript>Socks</subscript>
         <subscript>1</subscript>
      </parameterValue>
      <parameterValue value="3000">
```

```
      <subscript>Socks</subscript>
      <subscript>2</subscript>
    </parameterValue>
    <parameterValue value="9000">
      <subscript>Socks</subscript>
      <subscript>3</subscript>
    </parameterValue>
    <parameterValue value="15000">
      <subscript>Socks</subscript>
      <subscript>4</subscript>
    </parameterValue>
  </parameterValues>

  <!--Production cost for each product-->
  <parameterValues parameterId="ProductionCost">
    <parameterValue value="20.00">
      <subscript>Trousers</subscript>
    </parameterValue>
    <parameterValue value="21.00">
      <subscript>Shirts</subscript>
    </parameterValue>
    <parameterValue value="6.00">
      <subscript>Socks</subscript>
    </parameterValue>
  </parameterValues>

  <!--Rate of production for each product-->
  <parameterValues parameterId="ProductionRate">
    <parameterValue value="10">
      <subscript>Trousers</subscript>
    </parameterValue>
    <parameterValue value="9">
      <subscript>Shirts</subscript>
    </parameterValue>
    <parameterValue value="30">
      <subscript>Socks</subscript>
    </parameterValue>
  </parameterValues>

  <!--Number of working days availble in each period-->
  <parameterValues parameterId="WorkingDays">
    <parameterValue value="20">
      <subscript>1</subscript>
    </parameterValue>
    <parameterValue value="22">
      <subscript>2</subscript>
    </parameterValue>
    <parameterValue value="20">
      <subscript>3</subscript>
    </parameterValue>
    <parameterValue value="19">
      <subscript>4</subscript>
    </parameterValue>
  </parameterValues>
```

```
    <!--Storage Costs for each product-->
    <parameterValues parameterId="StorageCost">
      <parameterValue value="0.50">
        <subscript>Trousers</subscript>
      </parameterValue>
      <parameterValue value="0.50">
        <subscript>Shirts</subscript>
      </parameterValue>
      <parameterValue value="0.01">
        <subscript>Socks</subscript>
      </parameterValue>
    </parameterValues>
  </parameterData>
```

The value(s) for a single parameter is specified in the *"<parameterValues>"* element specifying a *'parameterId'* to relate it to the declaration. An individual value is specified using the *"<parameterValue>"* element. The value is encapsulated by the *'value'* attribute, which accepts floating point and integer values.

In the case where a parameter is not a single valued parameter i.e. is indexed over one or more sets, each value has to be associated with a unique collection of subscripts from the sets over which it is indexed. This is achieved using the *"<subscript>"* tag. The number of subscripts specified for each value is equivalent to the number of sets over which the parameter is indexed. The subscripts also have to be provided in the same order as the related *"<index>"* elements in the declaration. The reason for this restriction is that AML assumes that all indexed data is sparse, and as such requires that values are related to specific set subscripts.

3.3 Variables

Variables are declared within the *"<variables>"* node of the model, by means of the *"<variable>"* tag. Each variable has a unique name, which is given by the *'variableId'* attribute. A variable may be declared over one or more sets using the *"<index>"* tag. AML requires that a type is specified for each variable via the *"valueType"* attribute e.g. 'integer', 'binary', or 'real'. A value of 'integer' indicates that the variable may only take integer values, whereas a value of 'binary' indicates that the variable is binary i.e. may only take the values '0' and '1'. The value 'real' indicates that the variable is a free variable i.e. may assume real number values. The variables for the example model are declared as follows:

```
  <variables>
    <variable variableId="Produce" valueType="real">
      <index setId="product"/>
      <index setId="period"/>
    </variable>
    <variable variableId="Store" valueType="real">
      <index setId="product"/>
      <index setId="period"/>
    </variable>
    <variable variableId="Sell" valueType="real">
      <index setId="product"/>
      <index setId="period"/>
    </variable>
  </variables>
```

Although not necessary in the given example, it is also possible to specify bounds for variables using the *"<bound>"* tag within a variable declaration. Each bound has a *'boundValue'* attribute which

specifies the numeric limit for the variable, and a '*comparator'* attribute which specifies the bound equality or inequality condition e.g. 'equalTo', 'lessThan', 'greaterThan' etc. If the model requirements specified that more than one item of each manufactured product should be sold within each time period, a strict lower bound of 1 would have to be placed on the *Sell* variable. The corresponding declaration would be:

```
<variable variableId="Sell" valueType="floating">
  <index setId="product"/>
  <index setId="period"/>
  <bound comparator='greaterThan' boundValue='1.0'/>
</variable>
```

The '*<variable>'* tag may contain several '*<bound>'* tags, thus allowing the specification of multiple bounds for each variable, although in normal practice only two are required i.e. an upper and a lower bound.

3.4 Macros

A macro is a named encapsulation of an expression, which can be referred to in the model once declared as opposed to having to repeat the expression each time it is referenced/used. Macros are defined using the *"<macro>"* tag, which can occur, in the *"<macros>"* section of the model document. The macro name is specified via the *'macroId'* attribute.

The example model provides four declarations which are good candidates for encapsulation as macros, namely: *GrossRevenues; TotalProductionCost; TotalStorageCosts;* and TotalCosts. The corresponding macro definitions are given by:

```
<macros>
  <macro macroId="GrossRevenues">
    <function>
      <applySetFunction>
        <setFunction functionId="SUM">
          <index setId="product"/>
          <index setId="period"/>
        </setFunction>
        <function>
          <basicFunction>
            <lhs>
              <parameterReference parameterId="Price">
                <index setId="product"/>
              </parameterReference>
            </lhs>
            <operator>*</operator>
            <rhs>
              <variableReference variableId="Sell">
                <index setId="product"/>
                <index setId="period"/>
              </variableReference>
            </rhs>
          </basicFunction>
        </function>
      </applySetFunction>
    </function>
  </macro>

  <macro macroId="TotalProductionCosts">
```

```
  <function>
    <applySetFunction>
      <setFunction functionId="SUM">
        <index setId="product"/>
        <index setId="period"/>
      </setFunction>
      <function>
        <basicFunction>
          <lhs>
            <parameterReference parameterId="ProductionCost">
              <index setId="product"/>
            </parameterReference>
          </lhs>
          <operator>*</operator>
          <rhs>
            <variableReference variableId="Produce">
              <index setId="product"/>
              <index setId="period"/>
            </variableReference>
          </rhs>
        </basicFunction>
      </function>
    </applySetFunction>
  </function>
</macro>

<macro macroId="TotalStorageCosts">
  <function>
    <applySetFunction>
      <setFunction functionId="SUM">
        <index setId="product"/>
        <index setId="period"/>
      </setFunction>
      <function>
        <basicFunction>
          <lhs>
            <parameterReference parameterId="StorageCost">
              <index setId="product"/>
            </parameterReference>
          </lhs>
          <operator>*</operator>
          <rhs>
            <variableReference variableId="Store">
              <index setId="product"/>
              <index setId="period"/>
            </variableReference>
          </rhs>
        </basicFunction>
      </function>
    </applySetFunction>
  </function>
</macro>

<macro macroId="TotalCosts">
  <function>
```

```
        <basicFunction>
          <lhs>
            <macroCall macroId="TotalProductionCosts"/>
          </lhs>
          <operator>+</operator>
          <rhs>
            <macroCall macroId="TotalStorageCosts"/>
          </rhs>
        </basicFunction>
      </function>
    </macro>
  </macros>
```

The expression encapsulated by the macro is defined by a *"<function>"* element. More information on expressions is provided in section 3.7.

3.5 Objective

An AML model as already mentioned does not require an objective. This is because the AML syntax is designed to be flexible enough to represent constraint satisfaction problems. The current syntax does however place a limit of one on the number of objectives that can be declared in a model. This is probably not desirable from a multi-objective optimization point of view, but AML strives not to deviate too far from existing algebraic modelling languages so as to enable the easy translation of AML models into these formats.

The objective function is encapsulated by the "*<objective>"* tag. The objective function requires a unique identifier, which is supplied as the value of the *'objectiveId'* attribute. For the example model, the objective is given by:

```
<objective objectiveId="Profit" target="MAX">
  <function>
    <basicFunction>
      <lhs>
        <macroCall macroId="GrossRevenues"/>
      </lhs>
      <operator>-</operator>
      <rhs>
        <macroCall macroId="TotalCosts"/>
      </rhs>
    </basicFunction>
  </function>
</objective>
```

The *'target'* attribute provides a means of indicating to the target system the direction of the optimization i.e. minimization or maximization. In addition to the illustrated value of *'MAX'*, this attribute can also accept a value of *'MIN'*. The objective function expression is encapsulated by a "*<function>"* element, which is described in more detail in section 3.7.

3.6 Constraints

Constraints are declared within the *"<constraints>"* node of the model using the *"<constraint>"* tag. Each constraint must have a unique id, specified as the value of the '*constraintId'* attribute. It must also have a comparator that defines the constraint's equality or inequality condition, this is specified using the *'comparator'* attribute, which can accept inequalities or equalities such as *'equalTo'*, *'lessThan'*, *'greaterThanOrEqualTo'* etc. The constraint statements for the example model are provided in the following listing, where the bold font is used to highlight the multi-period inventory

constraint presented in equation (4), which in turn is included primarily as a means of illustrating the use of subscript-expressions in AML.

```
<constraints>
  <constraint constraintId="ProductionCapacity" comparator="lessThanOrEqualTo">
    <index setId="period"/>
    <function>
      <applySetFunction>
        <setFunction functionId="SUM">
          <index setId="product"/>
        </setFunction>
        <function>
          <basicFunction>
            <lhs>
              <variableReference variableId="Produce">
                <index setId="product"/>
                <index setId="period"/>
              </variableReference>
            </lhs>
            <operator>/</operator>
            <rhs>
              <parameterReference parameterId="ProductionRate">
                <index setId="product"/>
              </parameterReference>
            </rhs>
          </basicFunction>
        </function>
      </applySetFunction>
    </function>
    <constraintRhs>
      <function>
        <parameterReference parameterId="WorkingDays">
          <index setId="period"/>
        </parameterReference>
      </function>
    </constraintRhs>
  </constraint>

  <constraint constraintId="InventoryBalance" comparator="equalTo">
    <index setId="product"/>
    <index setId="period"/>
    <function>
      <basicFunction>
        <lhs>
          <variableReference variableId="Produce">
            <index setId="product"/>
            <index setId="period"/>
          </variableReference>
        </lhs>
        <operator>+</operator>
        <rhs>
          <variableReference variableId="Store">
            <index setId="product"/>
            <index setId="period">
              <subscriptExpression>
```

```
                    <operator>-</operator>
                    <numericLiteral>1</numericLiteral>
                  </subscriptExpression>
                </index>
              </variableReference>
            </rhs>
          </basicFunction>
        </function>
        <constraintRhs>
          <function>
            <basicFunction>
              <lhs>
                <variableReference variableId="Sell">
                  <index setId="product"/>
                  <index setId="period"/>
                </variableReference>
              </lhs>
              <operator>+</operator>
              <rhs>
                <variableReference variableId="Store">
                  <index setId="product"/>
                  <index setId="period"/>
                </variableReference>
              </rhs>
            </basicFunction>
          </function>
        </constraintRhs>
      </constraint>

      <constraint constraintId="MaxStorageCapacity" comparator="lessThanOrEqualTo">
        <index setId="period"/>
        <function>
          <applySetFunction>
            <setFunction functionId="SUM">
              <index setId="product"/>
            </setFunction>
            <function>
              <variableReference variableId="Store">
                <index setId="product"/>
                <index setId="period"/>
              </variableReference>
            </function>
          </applySetFunction>
        </function>
        <constraintRhs>
          <function>
            <parameterReference parameterId="StorageCapacity"/>
          </function>
        </constraintRhs>
      </constraint>

      <constraint constraintId="LimitSupply" comparator="lessThanOrEqualTo">
        <index setId="product"/>
        <index setId="period"/>
        <function>
```

```
        <variableReference variableId="Sell">
          <index setId="product"/>
          <index setId="period"/>
        </variableReference>
      </function>
      <constraintRhs>
        <function>
          <parameterReference parameterId="Demand">
            <index setId="product"/>
            <index setId="period"/>
          </parameterReference>
        </function>
      </constraintRhs>
    </constraint>

  </constraints>
```

It is clear from the declaration that a constraint element (definition) consists of three major sub-element groups: the indices; the constraint expression or the left side; and the right side expression.

It is often necessary to define a constraint over one or more sets, for example where the variable(s) being constrained are indexed over sets. In traditional modelling environments, such constraints are typically referred to as vector constraints. This can be replicated in AML by specifying one or more *"<index>"* elements at the start of the constraint definition.

The constraint expression is encapsulated by a *"<function>"* element and represents the left side of the constraint. To deviate slightly, the *"<function>"* element of the constraint *InventoryBalance* highlighted in bold font, illustrates an index subscript expression i.e. an expression which performs an arithmetic operation on the subscript implied by an *"<index>"* element. In the case of the example model we want to subtract 1 from the current subscript. It should be stated that this feature is supported in a different fashion in different optimization systems. To ensure consistency, the subscript value should be numeric, and an explicit value should be provided in the case where it is 1 i.e. the expression evaluates to zero. The latter of these may only be ignored where the target modelling system provides a known default for unspecified values.

The right hand side of the constraint is encapsulated by the *"<constraintRhs>"* element. This in turn contains a *"<function>"* element that encapsulates the right hand expression of the constraint. This enables the use of expressions of any supported degree of complexity or nesting on both sides of a constraint.

3.7 Expressions

Expressions are encapsulated by the *"<function>"* element. Elements of this type can be nested within each other to an arbitrary depth thus enabling the representation of complex algebraic expressions.

A given *"<function>"* element can contain one of the following types of elements:

- **Numeric literals:** These are specified using the *"<numericLiteral>"* tag, and can be either integers or floats, where the precision of each is dictated by the XSD specification. A basic example of an expression which evaluates to a number is that used in the definition of the parameter *StorageCapacity*.
- **Parameter references:** Parameters can be referenced using the *"<parameterReference>"* tag, supplying the name of the referenced parameter as the value of the *'parameterId'* attribute. In the case where a parameter is indexed over one or more sets, the indices must also be provided using the *"<index>"* tag, where *"<index>"*elements are listed in the same order

as in the referenced parameter's declaration. The reason for this is because the index tag in such a context forms a subscript expression, and as such provides the option of subscript-level arithmetic. Majority of the expressions declared in the example contain parameter references, and as such provide ample examples of how to use the *"<parameterReference>"* tag.

- **Macro Calls:** Pre-defined macros can be referenced using the *"<macroCall>"* element, specifying the name of the macro as the value of the *'macroId'* attribute. As already mentioned, macros cannot be referenced within *"<function>"* elements used in rendering expressions for calculating parameter values. The objective function in the given example i.e. *Profit* provides a simple example of how to reference macros.
- **Variable References:** Variables can be referenced using the *"<variableReference>"* element, specifying the name of the referenced variable as the value of the *'variableId'* attribute. Variable references cannot occur within expressions for calculating parameter values. A number of examples of variable references are provided in the macro, and constraint declarations in the example model. Like parameter references, variable references also have to include references to the set(s) over which the variable is defined, in the form of *"<index>"* tags.
- **Set subscript functions:** A subscript function represents an operation that is performed on the subscripts of a set. A typical example is the 'CARD' function, which indicates that the number of elements in the set should be calculated. Subscript functions are represented using the *"<subscriptFunction>"* tag, where the *'functionId'* attribute specifies the name of the function, and the *'setId'* attribute supplies the name of the set to whose subscripts the function is to be applied. A full list of supported functions is available from the AML schema. To give a trivial example of how to use a subscript function, the next section defines a parameter for the number of products made by the factory:

```
<parameter parameterId="ProductCatalogueSize">
  <function>
    < subscriptFunction setId='Product' functionId='CARD'>
  </function>
</parameter>
```

- **Arithmetic Functions:** Expressions can be built with pre-defined arithmetic functions such 'TAN', 'ABS', 'COS', by using the *"<applyMathFunction>"* element. This tag consists of two tags the first of which is the *"<mathFunction>"* tag which specifies the function to be referenced through its *'functionId'* attribute. The second tag can either be a parameter or variable reference, numeric literal, a set subscript function, a macro reference, or a nested function i.e. sub-expression. Variable and macro references cannot be used within this tag if it appears in the context of a parameter value expression. In simple terms, the *"<applyMathFunction>"* can be interpreted as: apply the function specified in *"<mathFunction>"*node to the following element. An example of how to use the *"<applyMathFunction>"* is:

```
<applyMathFunction>
  <mathFunction functionId="POWER">
      <numericalParameter>2</numericalParameter>
  </ mathFunction>
  <function>
    <variableReference variableId="Sell">
        <index setId="product"/>
        <index setId="period"/>
    </variableReference>
  </ function>
</applyMathFunction>
```

This expression simply means that the value of the *Sell* variable should be raised to the power of 2 i.e. squared.

- **Indexed Operations:** Quite often it is necessary to perform operations over indices such as summations. For example summing the values of a parameter or variable over the elements of the set(s) on which it is indexed. This is represented in AML by the use of the *"<applySetFunction>"* element. This is perhaps an ambiguous name, but it indicates that the constituent expression should be repeatedly evaluated for each member of the set. This tag contains two sub-tags. The first of which is a*"<setFunction>"* element which specifies the set operation to perform through its *'functionId'* attribute e.g. 'SUM'. The second tag can be one of the following: a parameter reference, a variable reference, a macro reference, a numeric literal, a set subscript function or an embedded function i.e. expression. In the case where this tag is used in the context of defining an expression for calculating the value(s) of a parameter, variable and macro references cannot be used anywhere in its descendant tree. A number of examples of the use of indexed operations are provided in the sample model.
- **Basic Expression:** A basic expression is differentiated from the other expression types by the fact that it has a left side, a right side and an operator between the two. An algebraic example is *a+b* where *a* is the left side and *b* is the right side. A basic expression is expressed by means of the *"<basicFunction>"* tag, which contains three sub-tags: *"<lhs>"*, *"<operator>"*, and *"<rhs>"*. The *"<lhs>"*, and *"<rhs>"* define the left and right sides of the expression respectively, and can contain parameter and variable references, subscript functions, macro calls, numeric literals, and nested functions i.e. sub-expressions. The same rules regarding the use of macro calls and variable references apply when the expression context is that of a parameter value calculation, as to other expression elements. The *"<operator>"* element specifies the arithmetic operator, and can take any of the following values: '*', '+', '-', '/'. A number of functions in the sample model including the objective function expression, make use of the basic function type, and as such provide ample examples on how the tag is used.

4 Implementation Decisions

AML is purely a representation format, and ultimately has to be processed within a software or programming language environment. This section aims to present the common implementation decisions, which have been made in order to ensure the completeness of the format, and the justification behind such decisions. These decisions can be grouped into two broad categories, namely validation and processing, and vendor extensions, although there are obvious dependencies between the two.

4.1 Validation and Processing

Whereas it is possible to replicate a graph structure in XML (specifically XML Schema i.e. XSD), using constructs such as *'key'* and *'keyref'* constructs in conjunction with the XSD *'ID'* type, the acyclical graph structure advocated by SML cannot easily be replicated in XML. This implies that there are two stages of validation required for an AML model. The logic for the first stage is embedded in the grammar of the language, thus removing the need for routines at the software layer to perform this function. This phase can be referred to as the syntactic validation phase, as it focuses purely on the syntax of an AML model. The second stage validates the structure or semantics of the model and is not easily achievable in XML, and as such has to be performed in the software layer. This stage of validation can however be considered optional if the receiving system decides to adopt a GIGO (Garbage In, Garbage Out) philosophy.

In the case where the target implementation does not make use of such a philosophy, it is important to provide an API to enable the easy processing of AML models, preferably based on the DOM and SAX API for processing XML. The OSCP [Ezechukwu and Maros, 2003] API specification can be considered as an abstraction of the processing and validation API required for the manipulation of AML models. The reference implementation of the API also provides a set of shared components built on JAXB (Java API for XML Binding), which implement common validation rules for AML models. It is possible to base further implementations on these components, and to prove this reference implementations for MPL, GAMS, and AMPL have been based on them.

The basic and common validation rules for AML models can be briefly summarised as follows:

i. **Prevention of Cyclical References:** This is perhaps the most important of validation constraints that should be placed on an AML model. In simple terminology, an element should not reference itself within its own definition i.e. it should not appear in its own calling sequence. This is applicable to macros and parameters. In essence a *"<macro>"* element cannot contain any descendant node of the type *"<macroCall>"* with the same value of the *'macroId'* attribute as itself. In the case of parameters, a *"<parameter>"* definition that contains a *"<function>"* child element cannot contain a descendant child node of the type *"<parameterReference>"* element with the same *'parameterId'* value as itself.

ii. **Index Binding:** This can be explained in simple terms as, all indices i.e. set references that appear in an expression must either occur in the declaration (right side) of the expression, or must occur within the context of an iterative set function, such as a summation. This is a restriction, which is very common in algebraic modelling languages, and whereas it isn't a strictly stated requirement of structured modelling, it is required for resolving references to indexed variables or parameters. In XML terminology, an *"<index>"* element that occurs within a *"<parameterReference>"*, or *"<variableReference>"* node must occur in the parent of the top-level *"<function>"* node or as a child of the *"<setFunction>"* node of an ancestor *"<function>"* element, or as a child of the current element's sibling *"<setFunction>"* element.

iii. **Data Validation:** The model data is specified in a separate document from the model, and as there is no known means of introducing cross validation between both documents, it is important to validate the model data on the software layer. This includes ensuring that members are specified for each set that is not defined as the result of a set operation, and for each parameter that does not have an associated expression for calculating its values. In the case where a parameter is indexed over one or more sets, each *"<parameterValue>"* element must also be validated to ensure that its count of *"<subscript>"* elements is equivalent to the number of *"<index>"* elements used in the parameter declaration. Also that the value of each *"<subscript>"* element is a valid member of the set identified by the *"<index>"* element which occurs at the same location in the parameter declaration.

It is plausible that individual end-points or receptors of AML models may choose to perform custom validation in addition to the basic validation defined above.

4.2 Vendor Extensions

AML attempts to maintain a close a mapping as possible to existing algebraic modelling languages, and its basic syntax is an abstraction of the structure of existing modelling languages, however there are a number of discrepancies between modelling languages not limited to but exemplified by the different rules for the processing of indices or sets. These discrepancies often occur because of the particular strengths of these modelling languages in representing particular classes/types of models. AML cannot provide structures to represent similar functions without compromising its neutrality or losing its focus on being an abstraction of modelling languages as opposed to a new modelling language. However, it would still be helpful to programmers and modellers if a method could be provided which would allow the strengths of specific languages to be exploited without compromising the structure of AML. Thankfully, the XML syntax and processing APIs provide a number of ways of doing this, some of them cruder than others. The following paragraph briefly describes some of the options available for extending the core AML syntax to cater for specific modelling languages or problems.

i. Source: This is probably the least elegant solution of all, but is probably suited to situations where the target user or organisation wishes to customize or extend the AML syntax for a highly specific application that cannot be catered for via any other means. AML is distributed in source format, following the principles of open source software, and as such, it is possible for users to alter its constituent source files.

ii. Processing Instructions: The XML syntax allows special instructions to be embedded in XML text. These allow the author of an XML document to dictate that a particular subroutine should be executed in relation to the portion of the XML document to which the instruction is related. Since AML is based on XML, it is possible to use this construct to dictate that certain portions of a model should be processed with a specific software library or utility. Processing instructions are very flexible in the sense that they are based on a plug and play philosophy. The structure of the document does not have to be altered; rather hints can be provided as to how to process it. There is no guarantee however that the hints would be heeded, especially if the referenced software library cannot be loaded.

5 Conclusion and Further Research

This paper introduces an algebraic mark-up language that is designed primarily for the purpose of enabling distributed optimization, and aiding the integration of optimization software into decision support applications. It has demonstrated that it is possible to abstract the structure of optimization models using a meta-language such as XML, and that it is possible to translate such an abstraction into existing formats, thereby reducing the coupling between disparate systems. It has also provided details of how the language can be customized for specific scenarios or target systems, in order to cater for requirements that are more specific to particular users. Future research on this language will focus on enhancing the core syntax to enable the expression of more complex models, particularly stochastic, and combinatorial optimization models. In addition to scheduled enhancements, it is very likely that user testing will introduce issues as yet unconsidered in the design of the language. Such issues will have to be reviewed on an individual basis, and may require enhancements to the syntax of the language or modifications to the validation routines.

6 Bibliography

1. *Algebraic Markup Language*, http://www.doc.ic.ac.uk/~oce/elsinore/aml.htm
2. H. K. Bhargava and R. Krishnan, *"The World Wide Web: Opportunities for Operations Research and Management Science"*, INFORMS Journal on Computing, 10:4, pp. 359-383, 1998
3. Obi C. Ezechukwu and Istvan Maros. *"OOF: Open Optimization Framework"*, DoC Departmental Technical Reports 2003/7, Imperial College, London, 2003
4. Obi C. Ezechukwu and Istvan Maros. *"OSCP: Optimization Service Connectivity Protocol"*, (forthcoming)
5. Obi C. Ezechukwu and Istvan Maros. *"ORML: Optimization Reporting Markup Language"*, (forthcoming)
6. Robert Fourer, David M. Gay, and Brian W. Kernighan. *"AMPL: A Modeling Language for Mathematical Programming"*, Duxbury Press/Brooks/Cole Publishing Company, 1993
7. Robert Fourer, and Jean-Pierre Goux, *"Optimization as an Internet Resource"*, INTERFACES 31:2, pp. 130-155, 2001
8. A.M. Geoffrion. *"An Introduction to Structured Modelling"*, Management Science, 33:547-588, 1987
9. Bjarni V. Halldorsson, Erlendur S. Thorsteinsson, Bjarni Kristjansson. *A "Modelling Interface to Non-Linear Programming Solvers. An Instance: xMPS, the extended MPS format"*, http://www.maximal-usa.com/papers/xmps/xmps.pdf, 2000
10. *JSolver*, http://www.ilog.com
11. Bjarni Kristjansson. *"Optimization Modeling in Distributed Applications: How New Technologies such as XML and SOAP allow OR to provide Web-based Services"*, http://www.maximal-usa.com/slides/Svna01Max/index.htm, 2001
12. *Mathematical Markup Language (MathML™) 1.01 Specification*, http://www.w3.org/TR/REC-MathML/
13. *MPL Modeling System*. http://www.maximal-usa.com/mpl/mplbroc.html
14. *MPL OptiMax 2000 Component Library*, http://www.maximal-usa.com
15. *MPS Input Format*, http://www-fp.mcs.anl.gov/otc/Guide/OptWeb/continuous/constrained/linearprog/mps.html
16. *OPL Studio*, http://www.ilog.com

17. *Simple Object Access Protocol (SOAP)*, http://www.w3.org/TR/SOAP/
18. *The AIMMS Modelling System*, http://www.aimms.com
19. *The IBM Optimization Subroutine Library (OSL)*, http://www.research.ibm.com/osl/
20. *The GAMS System*, http://www.gams.com/docs/intro.htm
21. *XML*, http://www.w3.org/XML/
22. *XML Schema*, http://www.w3.org/XML/Schema
23. *xMPS - The Extended MPS Format*, http://www.maximal-usa.com/papers/xmps/
24. *XSL*, http://www.w3.org/Style/XSL/

Appendix A. Multi-Period Production Planning Model

This section provides the AML model and data document listing for the example model presented in section 2.

1 Model Listing

```
<?xml version='1.0' encoding='windows-1252'?>
<aml:optimizationModel modelId="ProductionPlanning"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:aml="http://www.doc.ic.ac.uk/~oce/elsinore/2003/AML">

   <!--
      Basic production planning model.
   -->
   <sets>

      <set setId="clothing" alias="c"/>

      <set setId="accessories" alias="a"/>

      <set setId="product" alias="p">
         <setOperation   operationId="UNION"
                  leftSetId="clothing"
                  rightSetId="accessories"/>

      </set>
      <set setId="period" alias="q"/>
   </sets>

   <parameters>
      <parameter parameterId="Price">
         <index setId="product"/>
      </parameter>

      <parameter parameterId="Demand">
         <index setId="product"/>
         <index setId="period"/>
      </parameter>

      <parameter parameterId="ProductionCost">
         <index setId="product"/>
      </parameter>

      <parameter parameterId="ProductionRate">
         <index setId="product"/>
      </parameter>

      <parameter parameterId="WorkingDays">
         <index setId="period"/>
      </parameter>

      <parameter parameterId="StorageCost">
         <index setId="product"/>
```

```
    </parameter>

    <parameter parameterId="StorageCapacity">
      <function>
        <numericLiteral>800</numericLiteral>
      </function>
    </parameter>
  </parameters>

  <variables>
    <variable variableId="Produce" valueType="real">
      <index setId="product"/>
      <index setId="period"/>
    </variable>
    <variable variableId="Store" valueType="real">
      <index setId="product"/>
      <index setId="period"/>
    </variable>
    <variable variableId="Sell" valueType="real">
      <index setId="product"/>
      <index setId="period"/>
    </variable>
  </variables>

  <macros>
    <macro macroId="GrossRevenues">
      <function>
        <applySetFunction>
          <setFunction functionId="SUM">
            <index setId="product"/>
            <index setId="period"/>
          </setFunction>
          <function>
            <basicFunction>
              <lhs>
                <parameterReference parameterId="Price">
                  <index setId="product"/>
                </parameterReference>
              </lhs>
              <operator>*</operator>
              <rhs>
                <variableReference variableId="Sell">
                  <index setId="product"/>
                  <index setId="period"/>
                </variableReference>
              </rhs>
            </basicFunction>
          </function>
        </applySetFunction>
      </function>
    </macro>

    <macro macroId="TotalProductionCosts">
      <function>
        <applySetFunction>
```

```
        <setFunction functionId="SUM">
          <index setId="product"/>
          <index setId="period"/>
        </setFunction>
        <function>
          <basicFunction>
            <lhs>
              <parameterReference parameterId="ProductionCost">
                <index setId="product"/>
              </parameterReference>
            </lhs>
            <operator>*</operator>
            <rhs>
              <variableReference variableId="Produce">
                <index setId="product"/>
                <index setId="period"/>
              </variableReference>
            </rhs>
          </basicFunction>
        </function>
      </applySetFunction>
    </function>
</macro>

<macro macroId="TotalStorageCosts">
    <function>
      <applySetFunction>
        <setFunction functionId="SUM">
          <index setId="product"/>
          <index setId="period"/>
        </setFunction>
        <function>
          <basicFunction>
            <lhs>
              <parameterReference parameterId="StorageCost">
                <index setId="product"/>
              </parameterReference>
            </lhs>
            <operator>*</operator>
            <rhs>
              <variableReference variableId="Store">
                <index setId="product"/>
                <index setId="period"/>
              </variableReference>
            </rhs>
          </basicFunction>
        </function>
      </applySetFunction>
    </function>
</macro>

<macro macroId="TotalCosts">
    <function>
      <basicFunction>
        <lhs>
```

```
            <macroCall macroId="TotalProductionCosts"/>
          </lhs>
          <operator>+</operator>
          <rhs>
            <macroCall macroId="TotalStorageCosts"/>
          </rhs>
        </basicFunction>
      </function>
    </macro>
</macros>


<objective objectiveId="Profit" target="MAX">
    <function>
      <basicFunction>
        <lhs>
          <macroCall macroId="GrossRevenues"/>
        </lhs>
        <operator>-</operator>
        <rhs>
          <macroCall macroId="TotalCosts"/>
        </rhs>
      </basicFunction>
    </function>
</objective>

<constraints>
    <constraint constraintId="ProductionCapacity" comparator="lessThanOrEqualTo">
      <index setId="period"/>
      <function>
        <applySetFunction>
          <setFunction functionId="SUM">
            <index setId="product"/>
          </setFunction>
          <function>
            <basicFunction>
              <lhs>
                <variableReference variableId="Produce">
                  <index setId="product"/>
                  <index setId="period"/>
                </variableReference>
              </lhs>
              <operator>/</operator>
              <rhs>
                <parameterReference parameterId="ProductionRate">
                  <index setId="product"/>
                </parameterReference>
              </rhs>
            </basicFunction>
          </function>
        </applySetFunction>
      </function>
      <constraintRhs>
        <function>
          <parameterReference parameterId="WorkingDays">
```

```
            <index setId="period"/>
          </parameterReference>
        </function>
    </constraintRhs>
</constraint>

<constraint constraintId="InventoryBalance" comparator="equalTo">
    <index setId="product"/>
    <index setId="period"/>
    <function>
      <basicFunction>
        <lhs>
          <variableReference variableId="Produce">
            <index setId="product"/>
            <index setId="period"/>
          </variableReference>
        </lhs>
        <operator>+</operator>
        <rhs>
          <variableReference variableId="Store">
            <index setId="product"/>
            <index setId="period">
              <subscriptExpression>
                <operator>-</operator>
                <numericLiteral>1</numericLiteral>
              </subscriptExpression>
            </index>
          </variableReference>
        </rhs>
      </basicFunction>
    </function>
    <constraintRhs>
      <function>
        <basicFunction>
          <lhs>
            <variableReference variableId="Sell">
              <index setId="product"/>
              <index setId="period"/>
            </variableReference>
          </lhs>
          <operator>+</operator>
          <rhs>
            <variableReference variableId="Store">
              <index setId="product"/>
              <index setId="period"/>
            </variableReference>
          </rhs>
        </basicFunction>
      </function>
    </constraintRhs>
</constraint>

<constraint constraintId="MaxStorageCapacity" comparator="lessThanOrEqualTo">
    <index setId="period"/>
    <function>
```

```
            <applySetFunction>
               <setFunction functionId="SUM">
                  <index setId="product"/>
               </setFunction>
               <function>
                  <variableReference variableId="Store">
                     <index setId="product"/>
                     <index setId="period"/>
                  </variableReference>
               </function>
            </applySetFunction>
         </function>
         <constraintRhs>
            <function>
               <parameterReference parameterId="StorageCapacity"/>
            </function>
         </constraintRhs>
      </constraint>

      <constraint constraintId="LimitSupply" comparator="lessThanOrEqualTo">
         <index setId="product"/>
         <index setId="period"/>
         <function>
            <variableReference variableId="Sell">
               <index setId="product"/>
               <index setId="period"/>
            </variableReference>
         </function>
         <constraintRhs>
            <function>
               <parameterReference parameterId="Demand">
                  <index setId="product"/>
                  <index setId="period"/>
               </parameterReference>
            </function>
         </constraintRhs>
      </constraint>

   </constraints>
</aml:optimizationModel>
```

2 Data Listing

```
<?xml version='1.0' encoding='windows-1252'?>
<aml:optimizationModelData modelId="ProductionPlanning" modelInstanceId="Illustration"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:aml="http://www.doc.ic.ac.uk/~oce/elsinore/2003/AML">

   <setData>
      <!--products-->
      <setContents setId="clothing">
         <subscript>Trousers</subscript>
         <subscript>Shirts</subscript>
      </setContents>
      <setContents setId="accessories">
         <subscript>Socks</subscript>
```

```
    </setContents>

    <!--period idenifiers-->
    <setContents setId="period">
      <subscript>1</subscript>
      <subscript>2</subscript>
      <subscript>3</subscript>
      <subscript>4</subscript>
    </setContents>
  </setData>

  <parameterData>
    <!--product prices-->
    <parameterValues parameterId="Price">
      <parameterValue value="39.50">
        <subscript>Trousers</subscript>
      </parameterValue>
      <parameterValue value="35.00">
        <subscript>Shirts</subscript>
      </parameterValue>
      <parameterValue value="5.99">
        <subscript>Socks</subscript>
      </parameterValue>
    </parameterValues>

    <!--Demand for each product-->
    <parameterValues parameterId="Demand">
      <parameterValue value="1300">
        <subscript>Trousers</subscript>
        <subscript>1</subscript>
      </parameterValue>
      <parameterValue value="800">
        <subscript>Trousers</subscript>
        <subscript>2</subscript>
      </parameterValue>
      <parameterValue value="6000">
        <subscript>Trousers</subscript>
        <subscript>3</subscript>
      </parameterValue>
      <parameterValue value="3000">
        <subscript>Trousers</subscript>
        <subscript>4</subscript>
      </parameterValue>

      <parameterValue value="1750">
        <subscript>Shirts</subscript>
        <subscript>1</subscript>
      </parameterValue>
      <parameterValue value="1300">
        <subscript>Shirts</subscript>
        <subscript>2</subscript>
      </parameterValue>
      <parameterValue value="4000">
        <subscript>Shirts</subscript>
        <subscript>3</subscript>
```

```
    </parameterValue>
    <parameterValue value="8000">
      <subscript>Shirts</subscript>
      <subscript>4</subscript>
    </parameterValue>

    <parameterValue value="4000">
      <subscript>Socks</subscript>
      <subscript>1</subscript>
    </parameterValue>
    <parameterValue value="3000">
      <subscript>Socks</subscript>
      <subscript>2</subscript>
    </parameterValue>
    <parameterValue value="9000">
      <subscript>Socks</subscript>
      <subscript>3</subscript>
    </parameterValue>
    <parameterValue value="15000">
      <subscript>Socks</subscript>
      <subscript>4</subscript>
    </parameterValue>
</parameterValues>

<!--Production cost for each product-->
<parameterValues parameterId="ProductionCost">
    <parameterValue value="20.00">
      <subscript>Trousers</subscript>
    </parameterValue>
    <parameterValue value="21.00">
      <subscript>Shirts</subscript>
    </parameterValue>
    <parameterValue value="6.00">
      <subscript>Socks</subscript>
    </parameterValue>
</parameterValues>

<!--Rate of production for each product-->
<parameterValues parameterId="ProductionRate">
    <parameterValue value="10">
      <subscript>Trousers</subscript>
    </parameterValue>
    <parameterValue value="9">
      <subscript>Shirts</subscript>
    </parameterValue>
    <parameterValue value="30">
      <subscript>Socks</subscript>
    </parameterValue>
</parameterValues>

<!--Number of working days availble in each period-->
<parameterValues parameterId="WorkingDays">
    <parameterValue value="20">
      <subscript>1</subscript>
    </parameterValue>
```

```
            <parameterValue value="22">
               <subscript>2</subscript>
            </parameterValue>
            <parameterValue value="20">
               <subscript>3</subscript>
            </parameterValue>
            <parameterValue value="19">
               <subscript>4</subscript>
            </parameterValue>
         </parameterValues>

         <!--Storage Costs for each product-->
         <parameterValues parameterId="StorageCost">
            <parameterValue value="0.50">
               <subscript>Trousers</subscript>
            </parameterValue>
            <parameterValue value="0.50">
               <subscript>Shirts</subscript>
            </parameterValue>
            <parameterValue value="0.01">
               <subscript>Socks</subscript>
            </parameterValue>
         </parameterValues>
      </parameterData>
   </aml:optimizationModelData>
```