Index-Sets in Modeling Languages *

Tony Hürlimann [a]

[a] *Institute of Informatics, University of Fribourg, Switzerland*
E-mail: tony.huerlimann@unifr.ch , http://www-iiuf.unifr.ch/tcs/lpl

Index-sets are an integral and fundamental part of every mathematical modeling language. They assist the modeler in grouping various objects and entities. Index-sets are also used extensively in the mathematical notation in order to write an expression in a concise way. An example is the sigma notation for formulating the summation of an unknown positive number n of terms. In this paper, the concept of index-set is introduced in the context of modeling languages. The main objective is to propose an extension and generalization of the concept of index-sets, which is the concept of *hierarchical index-sets*. The paper concludes with an application.

Keywords: Modeling Language, Index-set, Hierarchical Structure

1. Introduction

Index-sets are an integral and fundamental part of every mathematical modeling language, such as AIMMS [**?**], AMPL [**?**], and LPL [**?**], in order to group various objects and entities. The same concept is also used in mathematical notation for indexing expressions, that is, to represent concisely a set of similar expressions. Indexing expressions is very common throughout mathematics. It is surprising how little has been published in the community of modeling language design on this concept. Even in mathematical textbooks, the indexed notation used in formula is often taken for granted and not much thought is given to it. A notable exception is [**?**] which devotes the whole Chapter 2 to indexed expressions and their use in mathematics.

All modeling languages in operations research have inherited two characteristics from matrix generators: they are *data-driven* and they are *index-driven*. Data-driven means that the data blocks and their dimension determine the data

* This research is supported by the Federal National Fond of Switzerland and financed by the project no. 1217-45922.95.

of the final instantiated model. The term "index-driven" has to be explained a little bit further. An *index* – from the point of view of matrix generators – is just a name of a row or a column in a block. The set of the names of all rows or columns is called *index-set*. Their cardinalities control the dimension of blocks. The Cartesian product of several index-sets defines a multi-dimensional block.

Index-sets are essential for mastering the complexity of large models. All elements in a model, such as variables, parameters, and constraints can appear in groups, in the same way they are indexed in the mathematical notation. As an example, let us consider a list of n products, each having a price (p), a volume (v), a name (n), a colour (c), or whatever properties are important to specify it. It is then common to define vectors p, v, n, and c expressing the list of properties as p_i, v_i, n_i, c_i, where $i \in \{1 \ldots n\}$. The notation p_i expresses the price, v_i the volume, n_i the name, and c_i the colour of the i-th product. The components of the different vectors need not necessary be numerical expression, they can be strings (n_i,), or elements of a discrete set ($c_i = \{\text{blue}, \text{green}, \text{red}, \text{black}, \ldots\}$). But all components of a vector must be elements of the same set: if a component is numerical, for example, then all other must also be numerical. Hence, the *type-set* of each vector needs to be specified as $p_i \in \mathbb{R}, v_i \in \mathbb{R}, n_i \in \mathbf{S}, c_i \in \mathbf{C}$ where $i \in \{1 \ldots n\}$. $\mathbb{R}$ being the set of reals, $\mathbf{S}$ being the set of all strings (of a given alphabet), and $\mathbf{C}$ being a finite set of colours. The part "$\{1 \ldots n\}$" is called *index-set*, i being the *index*. (Subsequently, if nothing is specified, the type-set is supposed to be $\mathbb{R}$.)

The concept of index-set used in algebraic languages is also closely related to the concepts of *entity set* and *key* in relational databases [**?**]. An attribute (or a set of attributes) whose values uniquely identify each entity in an entity set is called *key*, where *entity* is a "distinguishable thing" and a *set of entity* is a group of all similar entities [**?**]. In a company's database, an entity set, for example, consists of all employees, another of all products, etc. A main task in creating a database is to define the different entity sets and to assign a key (often an arbitrary serial number) to each entity; each entity set generates the different database *tables*. The keys are necessary in order to be able to link the different tables in an unequivocal way using the basic operations such as *selection*, *projection*, *join*, and others to generate *views* and other tables. It has been shown elsewhere by the means of a concrete example [**?**, p. 137–143], how the creation of a mathematical model with the help of an algebraic language corresponds to the creation of a database.

The example given above can be mapped directly to a table in a relational database: *Products* is the table name, its properties (or attributes) are *Price*, *Volume*, *Name*, and *Colour*. and one can write the table specification as:

Products(Name,Price,Volume,Colour)

If the *Name* is unique for each product, then it could be used as a key. The keys can be mapped to the elements in the index-set $\{1 \ldots n\}$. This shows that the manipulation of indexed entities in a modeling language can be based on a database calculus. The expressive power of a modeling language depends fundamentally on index-sets. Mapping between database tables and indexed parameters (or variables), and hence a clear separation between the structure and the instance of a model (its data) is only possible by making use of index-sets. At first sight, it seems that index-sets could be implemented by directly adopting the data structure of array, available in most programming languages. This is not the case! An index-set is a much more complex structure than an array, because of sparcity considerations. In modeling, it is common to declare large multi-dimensional tables containing ultimately only a small fractional part of entries. They are called sparse tables. Mapping such tables in a straightforward manner into arrays wastes precious memory. To manage efficiently (in time and space) such sparse tables is one of the most essential tasks of a modeling language.

This paper gives an overview of the concept of index-sets. Section 2 begins with an informal use of the indexed notation in mathematics. This is followed by a formal definition. Section 3 proposes a generalization of index-sets in modeling languages, which is called hierarchical index-set. This section gives also a unified view under which all kind of indexing structures can be subsumed. The last section presents a modeling example.

2. Definitions

The convention in algebraic notation to write a set of similar expression is to use *indexes* and *index-sets*. For example, a summation of the n (numerical) terms

$$a_1 + a_2 + \ldots a_{n-1} + a_n \tag{1}$$

is commonly abbreviated using the notation

$$\sum_{i=1}^{n} a_i \tag{2}$$

The expression (1) is called the *three-dots notation* and (2) is called the *sigma-notation*. It was introduced by Joseph Fourier in 1820, according to [**?**, p. 22]. Sometimes, the expression (2) is also written as

$$\sum_{1 \leq i \leq n} a_i \quad , \quad \sum_{i \in \{1 \ldots n\}} a_i \quad , \text{or} \quad \sum_{i \in I} a_i \quad \text{with} \quad I = \{1 \ldots n\} \tag{3}$$

The notations (3) are more general, because the index-set can be an arbitrary set defined even outside the summation expression.

Index-sets can also be calculated using the well known notation as follows:

$$\{i \in I \mid P(i)\} \tag{4}$$

where $P(i)$ is a property about i that is either true or false. The statement (4) expresses the set of all i in I such that the property $P(i)$ is true.

Every operator besides the $\sum$ operator which is associative and commutative can be used; common are $\prod$, $\bigcap$, $\bigcup$, $\bigvee$, $\bigwedge$, min, max, and others.

The index notation can be nested as in:

$$\bigwedge_{i \in \{1 \ldots m\}} \left(\sum_{j \in \{1 \ldots n\}} a_{i,j} x_j \right) \tag{5}$$

which represents m linear inequalities. This is normally written as

$$\sum_{j \in \{1 \ldots n\}} a_{i,j} x_j \quad \text{with} \quad i \in \{1 \ldots m\} \tag{6}$$

2.1. Simple Indexing Terms

More formally, let us call an *indexed notation* written as "$\bigodot_{i \in I} E_i$" a formal expression containing the following five elements: an *indexed operator* $\bigodot$, an *index-set* I, an *active index* i (in $i \in I$), a *passive index* i (in E_i) and an *indexed expression* E_i. The *index operator* may be any associative and commutative operator. The *index-set* is any countable (finite or infinite) collection of *elements*.

Definition: The *index-set* is any countable (finite or infinite) collection of *elements*, where an element is a undividable item called an *atom*. These index-set are called *simple*.

The *indexed expression* is an arbitrary expression. The *active index* is a name (or an identifier) representing an arbitrary element in the index-set. The *passive index* is also an identifier used in the indexed expression which must have the

same spelling as the active index. The term $i \in I$, consisting of the active index and the index-set, is called *indexing term*. The active index and the passive index are used as place-holders to define a bijective mapping between the index-set and a *set of indexed expressions*. Each element i in the index-set maps an indexed expression by replacing the passive index with the corresponding element i in the index-set. This mapping is called *index mechanism*. In fact, it defines the (bijective) function $f : i \xrightarrow{f} E_i$ with $i \in I$. This mapping supposes that the corresponding indexed expressions E_i exist. The *domain* of this function f is precisely the index-set, and the *codomain* (or the *range*) is the set of indexed expressions. The active index then is nothing else than an arbitrary name for the preimage i.

> **Definition:** The *indexed notation* is a formula expressing the fact that the *index operator* is applied to the set of *indexed expression* constructed by the *index mechanism*.

In the simplest case, the elements of the index-set are integers, but they can also be identifiers, strings or any other items representing elements. In the index-set $\{\text{spring}, \text{summer}, \text{autumn}, \text{winter}\}$, e.g., the four elements are identifiers.

2.2. Multiple Indexing Terms

Since the indexed notation is an arbitrary expression, the indexed expression can contain itself an indexed notation, which generates nested indexed notations, as in (7).

$$\bigodot_{i \in I} \left(\bigodot_{j \in J} E_{i,j} \right) \tag{7}$$

If both index operators are the same, then this can be expressed also by extending the syntax of an indexing term to a list of such terms. Let I and J be the two index-sets and i and j two active indices such that $i \in I$, $j \in J$ and let $E_{i,j}$ be an indexed expression with the two passive indices i and j mapping the two active indices i and j. Then the index notation of (7) can be written also as:

$$\bigodot_{i \in I, i \in J} E_{i,j} \tag{8}$$

The *indexing term* for this expression is now $\{i \in I, j \in J\}$.

One can also interpret the list of indexing term as a Cartesian product of index-sets. Hence, every index notation with a list of indexing terms can always be reduced to an index notation that contains only one index-set: the Cartesian product. In this case, of course, the elements of the index-set are tuples. This means that (8) can also be expressed as

$$\bigodot_{(i,j)\in I\times J} E_{i,j} \tag{9}$$

where the index-set is $I \times J$ and the active index is replaced by the notation (i, j) which is called *active index-tuple.* In this case, the names within the active index-tuples are called *active indices.* The active indices are now place-holders for the *components* of the tuple.

In some applications, there is no need to have access to the single components in a tuple. In such cases, an active index k may be used to represent an arbitrary *tuple* in the index-set as in:

$$\bigodot_{k\in I\times J} E_k \tag{10}$$

The index-set $I \times J$ is often replaced by a tuple set $K \subseteq I \times J$ in which case the expression (10) becomes

$$\bigodot_{k\in K} E_k \tag{11}$$

Now, we can see that indexed notation with multiple indexing terms really is reducible to indexed notation with one indexing term, we only need to extend the definition of index-sets:

Definition: **(redefinition)** The index-set is any countable (finite or infinite) collection of elements, where all elements are either undividable items called *atoms* or *tuples* containing all the same number of components. These index-sets are called *compound.*

It is interesting to note that the indexed notations (7)–(11) are essentially the same. However, (11) allows one to specify a proper *subset* of the full Cartesian product. This is important, because (11) has computational complexity of only $O = (\mid K \mid)$, whereas (9) has complexity of $O(\mid I \times J \mid)$.

Simetimes access to the tuple itself *and* to its components is needed. In such cases, one could use a combined syntax as in:

$$\bigodot_{k=(i,j)\in I\times J} (E_k, F_{i,j}) \tag{12}$$

where $(E_k, F_{i,j})$ is the indexed expression.

The previous considerations can be generalized to tuples of more than two components: If $I_1,\ldots,I_p$ are p non-empty, not necessary different index-sets containing only atoms then any subset of the Cartesian product $I_1 \times \ldots \times I_p$ is also an index-set, called *compound index-set*. Its elements are ordered tuples denoted by $(i_1,\ldots,i_p)$ with $i_1 \in I_1,\ldots,i_p \in I_p$. Each item i_k with $k \in \{1\ldots p\}$ in a tuple $(i_1,\ldots,i_p)$ is called a *component*. The integer p determines the *arity of the tuple*. All elements in a compound index-set have the same arity, and that the order of the components is significant. If the underlying index-sets $I_1,\ldots,I_p$ are all ordered then the order of the corresponding compound index-set is determined by the lexicographic ordering: the right most component is varied first.

The *active index-tuple* $(i_1,\ldots,i_p)$, contains the active indices of each component for the underlying index-sets. If an active index is needed for the tuple itself, it can be preceded by a new unique name as in $k = (i_1,\ldots,i_p)$, called *named active index-tuple*. k is now an active index for the tuple itself. This notation can be extended to any subset of components of the active index-tuple. For example, if an active index is used for a sub-tuple consisting of the second, fourth and p-th component besides an index for the tuple itself, one could write this as: "$k_1 = (i_2, i_4, i_p), k = (i_1,\ldots,i_p)$“.

We may call this a *list of named active index-tuple* (sorry). It is important to note that every name for a sub-tuple (the names before the equal signs) as well as all active indices in the *last* (complete) active index-tuple must be different from each other; however, all names in the *other* sub-tuples must occur in the last tuple, because they only define a selection from the last tuple. (Of course, the same selection may turn up more than once, defining several active indices for the same sub-tuple.)

The formula (8)–(11) can now be generalized to a list of p indexing terms as follows (where $K \subseteq I_1 \times \ldots \times I_p$):

$$\bigodot_{i_1\in I_1,\ldots,i_p\in I_p} E_{i_1,\ldots,i_p}\ , \qquad \bigodot_{(i_1,\ldots,i_p)\in I_1\times\ldots\times I_p} E_{i_1,\ldots,i_p}\ , \qquad \bigodot_{k\in I_1\times\ldots\times I_p} E_k\ , \qquad \bigodot_{k\in K} E_k \tag{13}$$

All tuples in the indexing terms have arity p. All four indexing expressions represent a p-dimensional table of expressions. *Simple index-sets* (which contain only atoms) can be interpreted as compound index-sets with arity 1. In this case, the parentheses around the "tuple" is not needed. The formula (12) can be generalized to

$$\bigodot_{\substack{h=(i_2,i_4),\\ l=(\ldots),\\ k=(i_1,\ldots,i_p)\in I_1\times\ldots\times I_p}} (D_h, E_k, F_l, G_{i_1,\ldots,i_p}) \tag{14}$$

2.3. Indexed Index-sets

Index-sets occurring in indexing terms can be themselves indexed. Let us define an index-set I and based on it an *indexed index-set* $J = \{J_i\}$ where J_i is an index-set and $i \in I$. The elements of J are index-sets themselves. As an example, suppose $I = \{a, b\}$, $J_a = \{1, 2\}$, and $J_b = \{2, 3\}$ then $J = \{\{1, 2\}, \{2, 3\}\}$. Based on this two sets I and J, one can form the following indexed notation:

$$\bigodot_{i\in I}\left(\bigodot_{j\in J_i} E_{i,j}\right) \quad \text{or} \quad \bigodot_{i\in I, j\in J_i} E_{i,j} \tag{15}$$

It is important to note, however, that the index i in $i \in I$ is an active index, whereas the i in J_i is a passive index. Now (15) can be interpreted as

$$\bigodot_{(i,j)\in K} E_{i,j} \quad \text{with} \quad K \subseteq I \times (J_1 \cup \ldots \cup J_{|I|}) \tag{16}$$

This means that the concept of indexed index-set can be reduced to compound index-set and does not add any new features. Nevertheless, a notation like (15) may sometimes be more convenient, because the tuples are presented as a hierarchical structure instead of a flat list.

2.4. Partioning a Active Index-tuple

A complication occurs in nested indexed notations in which the outer ($\bigodot$) and inner ($\bigotimes$) indexed operators are *different* as in

$$\bigodot_{i\in I}\left(\bigotimes_{j\in J} E_{i,j}\right) \tag{17}$$

because the components of compound index-set can be partitioned between the indexed operators. This occurs quite frequently in practical modeling. An exam-

ple is the flow conservation constraint in a transportation problem. Let I be a set of location, and $L \subseteq I \times I$ a set of links between the locations; furthermore, let $x_{i,j}$ with $(i,j) \in L$ be a flow at each link and a_i with $i \in I$ the supply (or demand) at each location, then the flow conservation constraint is often written as:

$$\bigwedge_{i \in I} \left(\sum_{j \in I} x_{i,j} - \sum_{j \in I} x_{j,i} = a_i \right) \tag{18}$$

However, (18) is only a correct formulation if $L \equiv I \times I$, but not if $L \subset I \times I$. In this later case, one must write

$$\bigwedge_{i \in I} \left(\sum_{j \in I | (i,j) \in L} x_{i,j} - \sum_{j \in I | (j,i) \in L} x_{j,i} = a_i \right) \tag{19}$$

Another slightly shorter syntax is more common and used in many textbooks of operations research, for example in [**?**, p. 64] and [**?**, p. 317]:

$$\bigwedge_{i \in I} \left(\sum_{j:(i,j) \in L} x_{i,j} - \sum_{j:(j,i) \in L} x_{j,i} = a_i \right) \tag{20}$$

From a syntactical point of view, (20) is not entirely satisfactory, because the two indexes in (i,j) are not symetrical. i is a passive and j is an active index. The same consideration holds for (j,i). The intention of the indexing term $j : (i,j) \in L$ is to construct an index-set which corresponds to a slice of L for a fixed element i, that is a *projection* of L into i. Since this is so common, an even shorter notation is proposed in this paper:

$$\{j \in L_{i,*}\} \tag{21}$$

where $L_{i,*}$ is called the *projection* of L into i. There are at least two advantages of this notation: It is short and the projection is more explicit. More importantly, it can be extended to any sub-tuple projection.

Definition: Let L be a compound index-set of arity p with $(i_1, \ldots, i_p) \in L$. A *projection* of L into a given subset K of its components of cardinality $p - q$ with fixed elemetns is another compound index-set of arity $q \leq p$ which is constructed from L as follows: (1) remove all tuples from L in which the components in K do not correspond to the fixed element in K; (2) remove the components in K from each remaining tuple (giving tuples of arity q); (3) remove duplicates from the tuple list.

Another advantage is that this notation is closely related to the notation of using *indexed index-sets* as in (15).

Example: Let $L = \{(1, a, B), (1, b, A), (2, b, A), (2, c, A), (3, c, A)\}$. Then the following projections can be constructed: $L_{1,*,*} = \{(a, B), (b, A)\}$, $L_{2,b,*} = \{A\}$, $L_{*,c,A} = \{2, 3\}$ and others. The union of all projections of L into $i \in I$ is called *total projection* of L into I. (Strictly speaking, the indexing term $\{i \in I\}$ in (19) or (20) should be replaced by $\{i \in I \mid \exists_j L_{i,j}\}$ (which is the total projection of L into I of the first index). This removes isolated nodes.)

The flow conservation constraint (20) can now be formulated using the proposed notation as follows:

$$\bigwedge_{i \in I} \left(\sum_{j \in L_{i,*}} x_{i,j} - \sum_{j \in L_{*,i}} x_{j,i} = a_i \right) \tag{22}$$

(It should be noted that using the proposed syntax for projection together with the introduction of named active tuple-lists makes the introduction of *labels* as proposed in [**?**, Chap. 2] unnecessary.)

In the algebraic language AMPL the following syntax is used for the flow conservation constraint [**?**, p. 86–87]:

```
C{i in I}: sum{(i,j) in L} x[i,j] - sum{(j,i) in L} x[j,i] = a[i];
```

which corresponds in the notation:

$$\bigwedge_{i \in I} \left(\sum_{(i,j) \in L} x_{i,j} - \sum_{(j,i) \in L} x_{j,i} = a_i \right) \tag{23}$$

This notation is at least misleading: The indexing term $(i, j) \in L$ suggests that the summation takes place over *all* tuples in L, which is not what is intended. The difficulty is that the indexing term cannot be interpreted independently from what occurs outside this term. We cannot deduce from the indexing term $(i, j) \in L$ which of the indices are already bound (passive indices) and which are not (active indices). This violates an important principle of decomposition in language design.

3. Hierarchical Index-Sets in Modeling Languages

Until now, we only considered index-sets which consist of tuples all having the same arity. If the arity is one, this reduces to simple index-sets in which all

elements are atoms. If the arity is greater than one we have compound index-sets. The concept of index-sets can be generalized to sets containing tuples with a varying number of components.

Definition: **(redefinition)** The index-set is any countable (finite or infinite) collection of elements, where the elements are atoms or tuples of *any* arity. These index-set are called *hierarchical*.This definition includes the two definitions of index-sets given before.

3.1. Syntactical Definition

We now give a precise syntax description of an index-set written in the extended Backus-Naur form. An *index-set* is a collection of elements. An *element* is an atom or a tuple. An *atom* is an undecomposable item which can be expressed by an identifier, a number, a string, a star, or an expression, which evaluates to a number or a string. (From the syntactical point of view an atom is just an expression or a star, since any identifier, number, or string is an expression.) The star is used for an unnamed component. A *tuple* is an ordered sequence of components. If $t_1, \ldots, t_p$ are components then a tuple can be expressed as $(t_1, \ldots, t_p)$. The number p is called the *arity* of the tuple. The components are listed from left to right, separated by commas and the tuple is surrounded by a left and a right parenthesis. This is the common *tuple-notation* to express an ordered list. A *component* can be tagged or untagged. An untagged component is called a *simple component*. A *tagged component* consists of a tag and a simple component separated by a colon. The *tag* is just an identifier, and a *simple component* is an element as defined above. The complete syntax of index-sets is given as follows:

```
IndexSet =       '{' Element {',' Element} '}'
Element =        Atom | Tuple
Atom =           Expression | '*'
Tuple =          '(' Component {',' Component} ')' |
                 Component {'.' Component}
Component =      [Tag] Element
Tag =            id ':'
```

Expression is a generic way to produce an atom whatever this is. A concrete way to define *Expression* may be to include all expressions which return a string of characters, an integer, or a symbolic name.

At first sight, it seems "unnatural" to introduce tuples as elements with a recursive structure; but tuples defined in this way are extremely useful as we shall see in a moment. If the concept of element would be restricted to atoms, index-sets would be little more than simple sets which could be implemented by arrays. Tuples make them much more complex and powerful. The historical background of tuples are existing algebraic languages, which clearly display their advantages. The modeling languages GAMS [?] and AMPL [?] introduced *compound index-sets*, and *indexed collections of index-sets*. LPL [?] introduced *hierarchical index-sets*. All three concepts can be smoothly unified and integrated using the notion of tuples. A compound index-set, is nothing more than an index-set, whose elements are all tuples with the *same* arity. An indexed collection of index-sets is also a compound index-set where a subset of components plays a dominating role. The most general index-sets are the hierarchical ones which consist of tuples with varying arity.

3.2. Semantic Definition

Hierarchical index-sets have a precise semantics. To define it, let us first introduce the two terms of *subsumable* and *trivially extended* tuples.

> **Definition:** A tuple Q of arity $q \leq p$ is call *subsumable* to another tuple P of arity p, if all components of Q are identical to the first q components of P. A tuple Q of arity $q \geq p$ *trivially extends* to another tuple P of arity p, if all components of P are identical to the first p components of P, and the last $q - p$ components are marked with a *star* (*).

Example: the tuple (a, b) is subsumable to the tuple (a, b, c). The tuple $(a, b, *)$ trivially extends the tuple (a, b).

The semantics of hierarchical index-sets (used in this paper) can be expressed by the following principle:

> **Axiom:** Two hierarchical index-sets I and J are *equivalent* if $I \backslash J$ only contains subsumable or trivially extended tuples of I or $J \backslash I$ only contains subsumable or trivially extended tuples of J.

From the axiom it follows that:

Theorem: A hierarchical index-set is an equivalence-class.

Proof: Every index-set I can be represented by a tuple-list, say J, obtained by removing all subsumable tuples from I and by removing all stared components at the end of a tuple. This follows from the symmetry and transitivity of the two terms *subsumable* and *trivially extended*. J is called the *canonical index-set*.

It should be noted that hierarchical index-sets are not ordinary sets, since (infinite) many ordinary tuple-sets are considered to be equivalent hierarchical index-sets. Nevertheless all set operators are well defined for index-sets. If $H(x)$ is the canonical representation of the index-set x, then it is easy to see that

$$\begin{aligned} H(A) \cap H(b) &\equiv H(A \cap B) \\ H(A) \cup H(b) &\equiv H(A \cup B) \\ H(A) \backslash H(b) &\equiv H(A \backslash B) \\ H(A) \times H(b) &\equiv H(A \times B) \end{aligned} \tag{24}$$

3.3. Different Representations of Index-sets

A hierarchical index-set (and hence every index-set) can be represented by a tree structure, called the *index-tree*. The index-tree is obtained in the following way from a canonical tuple-list: (1) draw a top-down path for every tuple, such that every component corresponds to a node, the nodes are *labelled* with the component; (2) connect the top of each path to an additional node called *root*; (3) merge two partial paths repeatedly from the root downwards as long as the nodes have the same *label*. The index-tree in which all partial paths are merged is called the *canonical index-tree*.

Conversely, it is interesting that every hierarchical structure can be transformed into a canonical index-set as defined above. This is done by first "unmerge“ all paths in the tree which represents the hierarchical structure, then removing the root and finally considering each path as a tuple.

Another way to represent a hierarchical index-set is a *set of index-sets*. The set of index-sets is obtained in the following way from the canonical index-tree: Traverse the tree in a depth-first manner, the siblings in any order. Before the first sibling generate "{“ and after the last sibling generate "}“. At each node write the label as a string to the output (except the star which does not generate

anything). This procedure will produce a nested string as output, which is a set of sets.

Example: the canonical index-set

```
{ (a,1), (a,2,X), b, (*,2), d }
```

can be represented as

```
{ a{1,2{X}}, b, {2}, d }
```

To capture the syntax of set of index-sets, only the non-terminal symbol `IndexSet` must be modified to:

```
IndexSet =        '{' ElementOrSet {',' ElementOrSet} '}'
ElementOrSet =    Element | [Label] IndexSet
Label =           Element
```

Conversely, the canonical tuple-list representation can be easily obtained by a set of index-set representation by parsing it without backtracking (and only *one* look-ahead symbol) (see [**?**]). Finally, removing all subsumable tuples and removing trivially extended components from each tuple generates the canonical index-list.

The described procedures transform a canonical index-list into a canonical index-tree, and further into a set of indexed-sets, and finally back into a canonical tuple-list, showing that the three representations are equivalent.

All this does not explain the benefits of *tags*. Tuples are – as we have seen – ordered list of components which reflect the levels in the index-tree. Tags have two benefits: the order of components within tuples can be permuted and, for compound sets, some components can, if necessary, even be dropped. Tags can also be used to concisely name partial path within the index-tree, or – in the index-set vocabulary – to name nested tuples. A third advantage of tags is that the *star* is not really needed even in noting the hierarchical index-set as tuple-list. This can be seen in the following notation (C representing the same index-set as before):

```
SET i := { a, b, d };   j := { 1, 2 };   k := { X, Y };
SET C{p:i,q:j,r:k}:= {(p:a,q:1), (p:a,q:2,r:X), p:b, q:2, p:d};
```

First, three simple index-sets (all elements are atoms) i, j, and k are defined. The hierarchical index-set C is indexed meaning that it is defined such that the first level can only have labels which are elements of i, the second level has only

labels which are elements of j, and the third level has labels which are elements of k. The indexing of index-sets, therefore, introduces domain checking. At the same time, the index-list introduces tags for the three levels in the index-tree.

Alternatively, one could also declare the first two levels as a compound set ij (with its tag pq within C) in the following way:

```
SET i := { a, b, d };   j := { 1, 2 };   k := { X, Y };
SET ij{p:i,q:j}:= { (p:a,q:1), (p:a,q:2) };
SET C{pq:ij,r:k}:={pq:(p:a,q:1),(pq:(p:a,q:2),r:X),p:b,q:2,p:d};
```

3.4. Operations on Index-sets

The main reasons why the concept of index-set has been generalized into a hierarchical structure are:

1. The concept is theoretical simple and all kind of indexing structure can be subsumed to this concept. It is the most general one.
2. Many applications can be put into a concise and straightforward representation (we shall see this in the next section).

To exploit the hierarchical structure of indexing, however, we need to introduces additional operators which allow to select items of the index-sets. In a hierarchical structure it is useful to access not only the elements (the paths from the root to the leaves), but also different subsets of *labels*, i.e. the nodes in a canonical index-tree.

Let i be a label of a node in an index-tree, then the following operations select a subset of nodes in the index-tree, (the root is labelled with the name of the index-set itself). The elements of the subsets are the labels:

$+i$: all direct descendants of node i,

$-i$: the (unique) parent of node i,

$\sim i$: all nodes which are descendants of i and which are not leaves,

$\&i$: all nodes which are descendants of i (including the leaves).

The unary operators "$-$", "$+$", "$\sim$", and " &" generate a simple index-set (all elements are atoms, the atoms being the labels of the nodes). Since the root is labelled with the name of the index-set itself, these operators can be applied to any name that represent an index-set: they are unary index-set operators. These operators have been introduced in my doctoral thesis [**?**, p. 22ff] and implemented

into the modeling language LPL in a very restricted form to generate hierarchical dependent constraints.

This section is concluded by giving a complete syntax description of the *indexing term* used in indexed notation for modeling languages.

```
IndexingTerm =    '{' Index {',' Index} ['|' LExpr] '}'
Index =           LNAIndex [AIndex IN] SetExpr
LNAIndex =        { id '=' AIndex ',' } [id '=']
AIndex =          id | '(' IdList ')'
IdList =          id { ',' id }
SetExpr =         SetUnaryOp SetFactor |
                  SetExpr {SetOp SetExpr} |
                  SetIndexOp IndexingTerm SetExpr |
                  IExpr ',' IExpr '..' IExpr
SetFactor =       id [AIndexList] | IndexSet | '(' SetExpr ')'
SetOp =           '+' | '*' | '-'
SetIndexOp =      AND | OR | PROJECT
SetUnaryOp =      '+' | '-' | '&' | '~'
AIndexList =      '[' PIndex {',' PIndex} ']'
PIndex =          EExpr | '*'
id =              <an identifier>
LExpr =           <an expression returning a Boolean value>
```

LNAIndex stands for a *list of named active index-tuple* as defined previously. *AIndex* stands for active indices, and *PIndex* for passive indices. *EExpr* is an expression returning an element in an index-set. The four unary operators are explained above. The binary operator AND, OR mean set intersection and set union, and PROJECT means total projection.

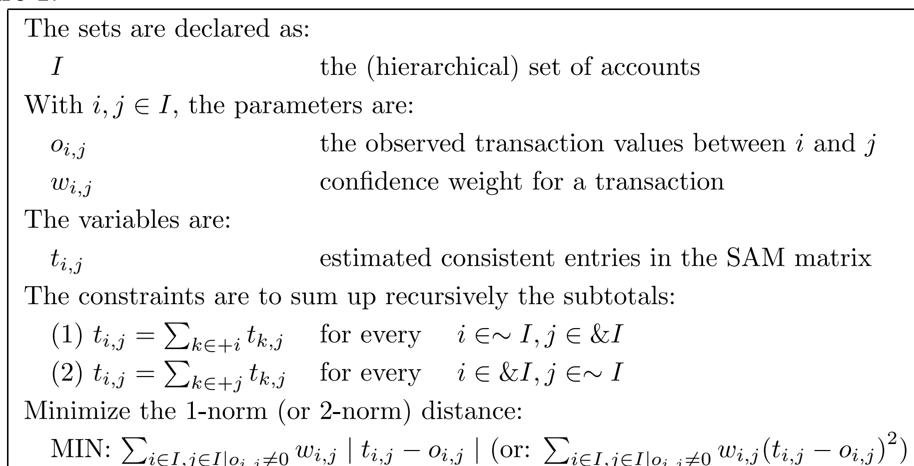
4. A Application in Modeling

The Social Accounting Matrix (SAM) is a two-dimensional, quadratic matrix in which economic transactions between the accounts are recorded. The rows and columns represent the accounts and an matrix-entry is a positive number representing the monetary value of transactions. The SAM is *consistent* if for each account the total row value corresponds to its total column value. The accounts often form a hierarchical structure. An example is given in KUI92.

The problem which has to be solved by a SAM estimation model can be formulated as follows:

Suppose a partially filled SAM is given. The data filled in were gathered through observation or other statistical means. In any case, it is very probable that these data are inconsistent. The problem is to replace them and complete the SAM in such a way that the resulting SAM is consistent and that a distance measure between the observations and the consistent entries becomes minimal.

The model can be formulated using the proposed set operators as given in Figure 1.



The sets are declared as:

I	the (hierarchical) set of accounts

With $i, j \in I$, the parameters are:

$o_{i,j}$	the observed transaction values between i and j
$w_{i,j}$	confidence weight for a transaction

The variables are:

$t_{i,j}$	estimated consistent entries in the SAM matrix

The constraints are to sum up recursively the subtotals:

(1) $t_{i,j} = \sum_{k \in +i} t_{k,j}$ for every $i \in \sim I, j \in \& I$

(2) $t_{i,j} = \sum_{k \in +j} t_{k,j}$ for every $i \in \& I, j \in \sim I$

Minimize the 1-norm (or 2-norm) distance:

MIN: $\sum_{i \in I, j \in I | o_{i,j} \neq 0} w_{i,j} \mid t_{i,j} - o_{i,j} \mid$ (or: $\sum_{i \in I, j \in I | o_{i,j} \neq 0} w_{i,j} (t_{i,j} - o_{i,j})^2$)

Figure 1: The SAM Model

Using a hierarchical index-set, the formulation of the model is very compact and readable (once the reader is familiar with the operators $\sim$, &, $-$, and $+$), and most importantly: its structure does not change with another accounting system.

5. Conclusion

A systematic overview of indexing mechanisms useful in modeling languages has been given in this paper. The concept of index-set has been extended and put into the unified framework of *hierarchical index-set*. An example was presented, which show the usefulness of this concept in practical modeling. Partial implementation into an older version of LPL has proven its feasibility. However, a conclusive judgement of whether the general concept is worth-while to be implemented into a modern modeling language has not been given. All examples, I have seen including the SAM model, *can* be done without using hierarchical

indexing structures employing only compound sets. Nevertheless, hierarchical indexing mechanism remains an intriguing concept and probably the last word has not been written about it.

References

[1] BISSCHOP J. and ENTRIKEN R., 1993. AIMMS, The Modeling System Paragon Decision Technology B.V.

[2] FOURER R., GAY D.M., KERNIGHAN B.W., 1993. AMPL, A Modeling Language For Mathematical Programming The Scientific Press, San Francisco.

[3] GEOFFRION A.M., 1992. Indexing in Modeling Languages for Mathematical Programming *Management Science*, 38(3), p. 325–344.

[4] GRAHAM R.L. and KNUTH D.E. and PATASHNIK O., 1994. Concrete Mathematics, A Foundation for Computer Science, second edition Addison–Wesley Publ., Reading, Massachusetts.

[5] HÜRLIMANN T., 1998. Hierarchical Index-sets in Modeling Languages, Working Paper, March 1998, Institute of Informatics, University of Fribourg, (a PostScript version is at the LPL site: ftp://ftp- iiuf.unifr.ch /pub/lpl/doc, file APMOD2.ps).

[6] HÜRLIMANN T., 1997. Computer-Based Mathematical Modeling, Habilitation Script, accepted by the Faculty of Economic and Social Sciences of the University of Fribourg, Switzerland, December 1997 Institute of Informatics, University of Fribourg.

[7] HÜRLIMANN T., 1997. Reference Manual for the LPL Modeling Language, Working Paper, Version 4.25, November 1997, Institute of Informatics, University of Fribourg, (newest version is always at the LPL-site: ftp://ftp-iiuf.unifr.ch /pub/lpl/doc, file Manual.ps).

[8] HÜRLIMANN T., 1987. LPL: A Structured Language for Modeling Linear Programs, Disseration, Peter Lang, Bern.

[9] IGNIZIO J.P. and CAVALIER T.M., 1994. Linear Programming, Prentice Hall, Englewood Cliffs, NJ.

[10] KUIP C.A.C., 1992, Index Sets in Mathematical Programming Modeling Languages, Proefschrift, Universiteit Twente, Netherland.

[11] MEERAUS A., 1983. An algebraic approach to modeling. *Journal of Economic Dynamics and Control*, 5, p.81–108.

[12] MITRA G., LUCAS C., MOODY S., 1995. Sets and Indices in Linear Programming Modelling and their Integration with Relational Data Models, *Computational Optimization and Applications*, 4, p. 263–283.

[13] NEMHAUSER G.L. and WOLSEY L.A., 1988. Integer and Combinatorial Optimization, Wiley.

[14] ULLMAN J.D., 1982. Principles of Database Systems, Computer Science Press.