Powerful modeling using array variables

Magne Myrtveit
ModellData AS
N-5120 Manger, Norway
Tel: +47 5637 4009
Fax: +47 5637 3500
E-mail: magmy@modeld.no

Abstract

As the field of system dynamics modeling is expanding, there is a continuos need for improvements of the available tools for developing simulation models. Lack of features like array variables often lead modelers to choose third generation languages like C when developing large, realistic models.

This paper describes the array variables of the POWERSIM language. Comparisons are made to other notations, including mathematics and DYNAMO. Index variables, array dimensions, subscripts, and functions operating on arrays are described. An important feature of POWERSIM is that the array notation goes well together with standard accumulator-flow diagrams (AFD) and causal loop diagrams used by system dynamicists. This makes the use of array variables almost as easy and intuitive as using scalars.

The transition from single scalar values to multi-element array variables is visualized through examples. Examples include capital stock with machines and buildings, work force with inexperienced and experienced workers, delay structures programmed as arrays, etc.

The array feature of POWERSIM has been used with success in several large-scale projects. Many modeling problems are not practically solvable without using arrays. Even models that can be developed using only scalars, sometimes become much easier to develop, explain and maintain when using arrays. In conclusion, the family of simulation problems that are best solved using a system dynamics tool, has been extended significantly through POWERSIM's array mechanism.

Powerful modeling using array variables

The array features of POWERSIM

Introduction

A dynamic model is a mathematical description of a real or imaginary system. The use of accumulator-flow models makes the power of mathematical integration available in an intuitive and straight-forward way, also for non-mathematicians. The use of vectors and matrixes (arrays) is a powerful feature of mathematics. Several approaches have been made to add array variables to dynamic modeling languages. A major weakness of the early approaches seems to be that the solutions are selected with limited focus on graphical editing of the simulation model as an AFD[1].

Our goal has been to provide the power of dimensioned variables (arrays) to POWERSIM's simulation language in such a way that both the textual and the graphical representation of models are kept as simple and intuitive as possible.

The inclusion of array variables has implications on three aspects of the simulation tool; 1) the textual simulation language (the equations); 2) the graphical accumulator-flow diagram; and 3) the user interface. Each of these will be discussed below. The following terms will be (informally) defined: array, scalar, literal, range, subrange, enumeration, dimension, subscript, index, constraint, and guard.

Scalar variables

Normal variables are called *scalar* variables, i.e., variables holding single numerical values. As an example, the definition[2]

```
const     InterestRate = 5%
```

defines[3] a scalar constant named *InterestRate* with the single value of 0.05. Scalars are displayed with a single frame in the diagram, as shown in Figure 1-a.

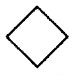
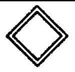
a) Scalar constant	b) Array constant
	

Figure 1: Scalar and array constant symbols

Array variables, array elements, vectors, and array literals

An *array* variable may[4] hold several distinct values (at the same time). Each distinct value is called an array *element*. As an example, the definition

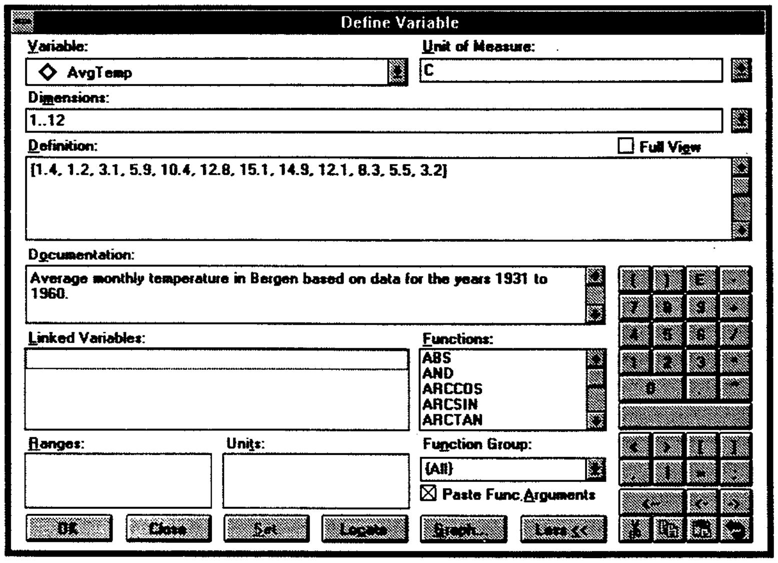
```
const     AvgTemp(1..12) = [1.4, 1.2, 3.1, 5.9, 10.4, 12.8, 15.1, 14.9, 12.1, 8.3, 5.5, 3.2]
```

defines a vector (one-dimensional array) with 12 elements numbered 1 through 12, inclusive. Each element holds an average monthly temperature[5] in Bergen, measured in °C.

[1]AFD – Accumulator Flow Diagram
[2]The definitions below are copied from the Equations view of POWERSIM.
[3]% is a POWERSIM postfix operator dividing its operand by 100, i.e., 5% = 5/100 = 0.05.
[4]It is possible to define arrays with only one element.
[5]Average temperatures 1931-60, Florida Bergen, Source: Det norske meteorologiske institutt.

A list of numbers enclosed in square brackets ([]) is used to define a vector *literal*. In the above example, the expression [1.4, 1.2, 3.1, 5.9, 10.4, 12.8, 15.1, 14.9, 12.1, 8.3, 5.5, 3.2] is a vector literal with 12 elements[6].

The dialog box for defining variables is the same both for scalars and arrays (see Figure 2). For an array, the Dimensions section is used to define each dimension of the variable. In our example, *AvgTemp* has one dimension, with elements 1 through 12.



Variable: AvgTemp

Unit of Measure: C

Dimensions: 1..12

Definition: [1.4, 1.2, 3.1, 5.9, 10.4, 12.8, 15.1, 14.9, 12.1, 8.3, 5.5, 3.2]

Documentation: Average monthly temperature in Bergen based on data for the years 1931 to 1960.

Figure 2: Dialog box for defining variables

A dimension must include a *range* specifying the element names of the dimension. In Figure 2 the text 1..12 in the Dimensions field defines a numerical subrange from 1 to 12, inclusive. Arrays are displayed using double lines in the diagram, as shown in Figure 1-b. The array element names and element values of *AvgTemp* are visualized below.

Value	1.4	1.2	3.1	5.9	10.4	12.8	15.1	14.9	12.1	8.3	5.5	3.2
Name	1	2	3	4	5	6	7	8	9	10	11	12

Functions operating on arrays

POWERSIM has several functions that take array arguments. As an example, the average temperature during one year may be computed using the ARRAVG function with *AvgTemp* as an argument, like this:

aux AvgTempYear = ARRAVG(AvgTemp)

The right hand side of the above equation may also be written either as *ARRAVG(AvgTemp(*))* or as *ARRAVG(AvgTemp(1..12))*.

Vectors, like *AvgTemp*, may be used, e.g., as arguments to the GRAPH family of functions. As an example, the expected temperature at a given month may be obtained using the following definition[7]:

aux ExpectedTemp = GRAPHCURVE(TIME MOD 12, 0, 1, AvgTemp)

[6]A single decimal number is called a *scalar literal*, e.g. 3.14.
[7]The TIME function returns the current time during the simulation. In this example it is assumed that the simulation time unit is months. The MOD operator computes the remainder after division. TIME MOD 12 will produce the values 0, 1, 2, 3..., 11, 0, 1, 2, 3, 4 for TIME values of 0 through 16.

Array subscripts for accessing array elements

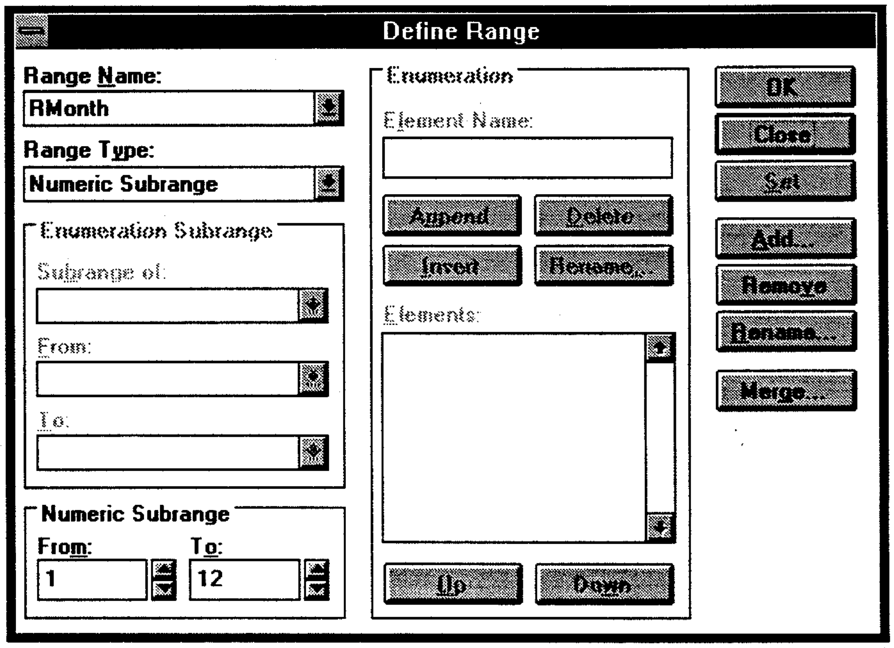
Individual array elements or ranges of elements may be accessed using array *subscripts*. As an example, the expression *AvgTemp(1)* obtains the average temperature in January, i.e., 1.4°C. Subscripts may also be used to identify sub-arrays. The expression *ARRAVG(AvgTemp(6..8))* computes the average temperature during the summer months.

Numeric subranges

Sometimes, it is useful to employ named sets of elements (ranges) when defining variable dimensions. One reason may be increased readability. Another, more important, reason is that changes to a named range definition will have effect on all variables that use the range. In the above example we may define the numerical subrange 1..12 as a range with the name *RMonth*.

```
range     RMonth = 1..12
```

This may be done in POWERSIM by using the Edit Define Range dialog box (see Figure 3).



Range Name: RMonth
Range Type: Numeric Subrange
Numeric Subrange From: 1 To: 12

Figure 3: Defining a range

Our definition of *AvgTemp* may now be changed to the following:

```
const     AvgTemp(RMonth) = [1.4, 1.2, 3.1, 5.9, 10.4, 12.8, 15.1, 14.9, 12.1, 8.3, 5.5, 3.2]
```

By expanding the Define Variable dialog box (using the More>> button, see Figure 2) we see that *RMonth* is presented in the list of Ranges (lower, left hand corner).

By selecting the entire text of the Dimensions field and double-clicking on *RMonth* in the Ranges list box, *RMonth* will be pasted into the Dimensions field of the Define Variable dialog box (see Figure 2).

Enumerated ranges

Elements of a range may be given mnemonic names. As an example, let us redefine the *RMonth* range as an enumeration of month names. This is done by opening the Define Range dialog box, and changing the Range Type to Enumeration. Element names are entered in the Elements list box by typing one Element Name at the time and then pressing the Append button (see Figure 3). The resulting range looks like this in equations view:

range RMonth = January, February, March, April, May, June, July, August, September, October, November, December

The temperature in January may now be expressed as *AvgTemp(January)*, and the average temperature in the summer months is *ARRAVG(AvgTemp(June..August))*.

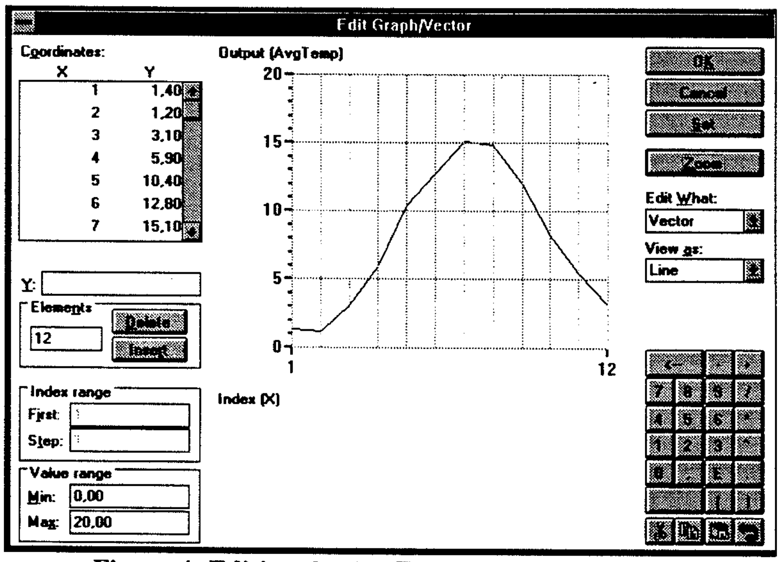
Enumeration subranges

It is also possible to define sub-ranges based on previously defined ranges. The four seasons may, as an example, be defined as sub-ranges of the *RMonth* range. This is done by defining a new range and setting Range Type to Enumeration Subrange in the Define Range dialog box (see Figure 3). By entering the name *RSummer*, and selecting *June* as From and *August* as To, the following range will be defined:

range RSummer = June..August

Editing vectors graphically using mouse and keyboard

Vectors and the GRAPH family of functions may be edited using the Edit Graph/Vector dialog box, which is opened by clicking the Graph button of the Define Variable dialog box (Figure 2). In Figure 4, the definition of *AvgTemp* is edited graphically. In the graph window, the mouse may be used to draw the graph. The list of vector elements can be edited by making selections in the Coordinates list box and typing in new values in the Y text box, inserting elements with Insert, deleting elements with Delete. Elements may also be copied or pasted via the clipboard using the Copy and Paste buttons (see the bottom row of the keypad in the dialog box).



Coordinates:

X	Y
1	1,40
2	1,20
3	3,10
4	5,90
5	10,40
6	12,80
7	15,10

Y:

Elements: 12

Index range
First:
Step:

Value range
Min: 0,00
Max: 20,00

Edit What: Vector

View as: Line

Figure 4: Editing the AvgTemp vector graphically

The average summer temperature was defined above based on the anonymous subrange *June..August*, like this: *ARRAVG(AvgTemp(June..August))*. Using the named subrange *RSummer*, we have the following alternative expression: *ARRAVG(AvgTemp(RSummer))*.

Index variables

New array variables can be defined based on anonymous ranges, named ranges, and range elements. As an example, a variable *AvgSummerTemp* may be defined as a vector with three elements; *June*, *July*, and *August*. This will be expressed either as

```
    aux      AvgSummerTemp(m=June..August) = AvgTemp(m)
or as
    aux      AvgSummerTemp(m=RSummer) = AvgTemp(m)
```

Observe the introduction of the *index* variable *m*, which is defined in the Dimensions section of the Define Variable dialog box (Figure 2), and used in the subscript of the expression defining the variable. In the examples above, the use of an index variable is required, as *AvgSummerTemp* and *AvgTemp* have different dimensions (see below).

AvgTemp:

1.4	1.2	3.1	5.9	10.4	12.8	15.1	14.9	12.1	8.3	5.5	3.2
Janu-ary	Febru-ary	March	April	May	June	July	Augus t	Sept-ember	Oct-ober	Nov-ember	Dec-ember

AvgSummerTemp:

June	July	Augus t

Using the index variable *m*, we specify which elements of *AvgTemp* are going to be used when defining *AvgSummerTemp*. Index names (like *Month* in the definition above) are local to each variable definition, and hence, the same name may be used in different variable definitions.

Implicit subscripts and indices

When dimensions match, index variables may be omitted, provided that indices of the elements all correspond on a one-to-one basis. Above, we've defined *AvgTemp* like this:

```
    const    AvgTemp(RMonth) = [1.4, 1.2, 3.1, 5.9, 10.4, 12.8, 15.1, 14.9, 12.1, 8.3, 5.5, 3.2]
```

The full form of the above definition is:

```
    const    AvgTemp(Month=RMonth) = [1.4, 1.2, 3.1, 5.9, 10.4, 12.8, 15.1, 14.9, 12.1, 8.3, 5.5,
                    3.2](Month)
```

Using arrays as parameters to dynamic objects

Array elements may be used in dynamic objects (e.g., time graph, time table, bar), as displayed in Figure 5. Here, a POWERSIM Bar object is used to display the values of the *AvgTemp* array elements. The labels listed to the left in the figure may be changed by the modeler. The slider bars[8] to the right in the figure show the current values of the array elements, and may also be used to change the values using the mouse.

[8]Optional scales may be added to the sliders.

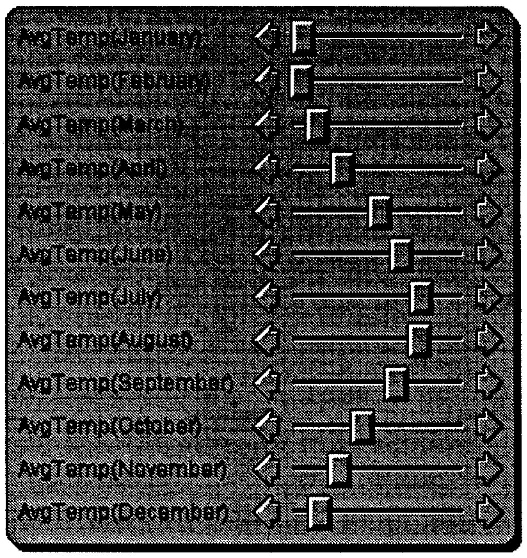


Figure 5: Accessing array elements from dynamic objects

Automatic display of variable values

During simulation, the value of a variable may be displayed in the statusbar by pressing the right mouse button while pointing at the variable. For array variables, the values of array elements will be displayed, separated by comma and enclosed in square brackets (as for array literals). An auto report of an array variable will display the first element of the variable.

Erroneous flows and edited flows

As with scalar variables, a variable definition may become invalid for various reasons (e.g., the use of unknown variables or wrong use of functions). POWERSIM will then display a question mark inside the variable symbol (see Figure 6-a). With the introduction of arrays, flows may also become invalid. E.g., if a flow and the connected level have different dimensions, the system may be unable to determine how the elements of the flow are going to be paired to the elements of the associated level. It is easy to come up with useful examples where the dimensions of connected flows and levels vary. Therefore, POWERSIM allows the modeler to edit the flow definition of levels. This is done by selecting the Flow radio button in the Define Variable dialog box, and editing the contents of the Definition text box. Hollow flow arrows are, by definition, used to represent conserved flows. POWERSIM will put a warning exclamation mark (!) on a flow if it has been changed from its default definition (see Figure 6-b). If there is an error in a flow definition, POWERSIM will display a question mark on the flow arrow (see Figure 6-c).

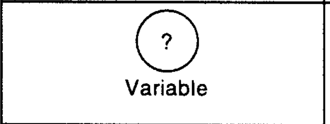
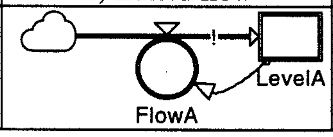
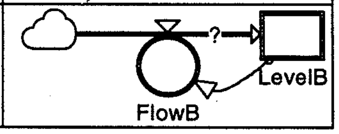
a) Error in definition	b) Edited flow	c) Error in flow
		

Figure 6: Errors and warnings in the diagram

The problem of validating array subscripts

It has been a major design goal for the array aspect of the POWERSIM language to provide full range check on array subscripts at the time of model construction (vs. run-time). This has two major advantages:

1. The model will run more efficiently (no time is used by the processor for range checking)
2. Simulation will never enter a state where array subscripts get out of bound.

As an illustration, let us assume that we have two vectors, *A* and *B*, with equal dimensions, and that each element of *B* is to be set equal to the element subsequent to the corresponding element of *A*, i.e., $B(i) = A(i+1)$. This will be OK, except for the last index *i*, were the subscript $(i+1)$ will be out of bounds.

We may attempt to solve the problem by means of an IF function, i.e., $B(i) = IF(i < LAST(i), A(i+1), 0)$. This equation will access $A(i+1)$ only for *i*'s in the range $FIRST(i)$ to $LAST(i)-1$. The only problem is that this solution, in general, will require run-time checking of array subscripts, as any expression may be used as the controlling condition of an IF function.

Constraints and guards

Therefore, in designing the POWERSIM language I chose to construct a separate conditional statement for constraining expressions based upon the values of index variables. A *constrained expression* is an expression followed by a constraint. A *constraint* is basically an index expression determining when the preceding expression should be evaluated. Below is an example where the problem presented above is solved using constrained expressions[9]:

B(i) = A(i+1) WHEN i < LAST(i)[10] BUT 0 WHEN DEFAULT

Here, two constrained expressions, separated by BUT are used to define *B*. The expressions are:

A(i+1) WHEN i < LAST(i)

and

0 WHEN DEFAULT

A boolean expression following WHEN is called a *guard.* DEFAULT is a guard that may be used in the last constrained expression to cover all cases that are not included in earlier guards in the constrained expression list.

The key to understanding the impact of constrained expressions on compile-time validation of array subscripts lies in the fact that only index variables, range elements, ranges and integer numbers are allowed as operands when expressing guards, array dimensions, and array subscripts. (Note that model variables are excluded from this list.) This implies that POWERSIM is able to determine and verify the values of all array subscripts without having to simulate the model.

Examples of when to use arrays

Replicated structures

Models often contain many identical or almost identical structures – especially large-scale models of realistic systems. If we, as an example, want to make a business simulator involving work force, capital, production, markets, etc., it is easy to identify several duplicate structures, e.g.:

1. Different kinds of capital, e.g. buildings and machines
2. Different market segments
3. Different kinds of workers
4. Different product lines
5. Different accounts payable, e.g., marketing expenses, goods, taxes, wages, dividends, debt

[9]An alternative way to write B(i) = A(i+1) WHEN i < LAST(i) BUT 0 WHEN DEFAULT is the following: B(i) = A(i+1) | i < LAST(i) ; 0 | DEFAULT

[10]The built-in function LAST computes the upper limit of a range or index variable. POWERSIM also has a FIRST function for returning the lower limit of a range or index variable. The COUNT function may be used to compute the number of elements in a range or separate values taken on by an index variable

Say, we want to make a model of four competing companies. This would involve four similar structures – the main differences being values of parameters (constants) and accumulators (levels). Once a model of one company is developed, models of more companies can be added just by adding a company dimension to every variable of the model.

There are two important advantages of expressing similar structures in the form of arrays:

1. Maintainability – Changes to an array structures immediately takes effect for all elements of the array.
2. Complexity – Models sometimes become significantly smaller when using arrays (see Figure 7).

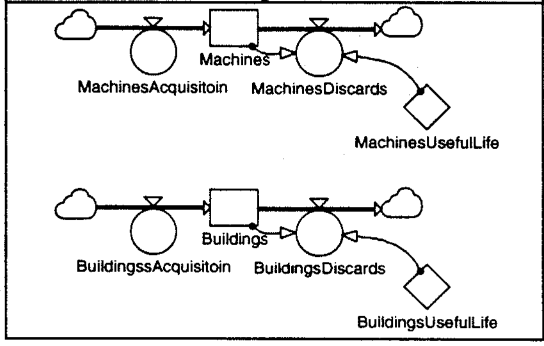
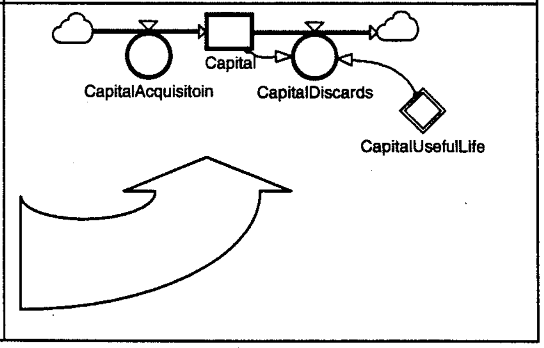
a) Separate structures for each type of capital	b) Structure with variables of capital type dimension
 Machines MachinesAcquisitoin MachinesDiscards MachinesUsefulLife Buildings BuildingssAcquisitoin BuildingsDiscards BuildingsUsefulLife	 Capital CapitalAcquisitoin CapitalDiscards CapitalUsefulLife

Figure 7: Combining similar scalar structures into one array structure

State transitions

When modeling systems involving state transitions, it is common to express this as a sequence of accumulators with interconnected flows. Some examples are:

1. Training of people, moving people from a less skilled to a more skilled state
2. Aging of people, moving people from one age group to the next
3. Production, moving goods from less finished to more finished states
4. Transport, moving goods from one place to another
5. Capital upgrade or maintenance, moving capital from one state to another

Figure 8-a displays a typical model of a production process using scalar variables. In Figure 8-b the same model is presented using vectors. The *Production* variable has one element less than *Goods* (observe the exclamation marks on the flows). Note that the structure of the array version will be unchanged, even if the number of production phases should be changed.

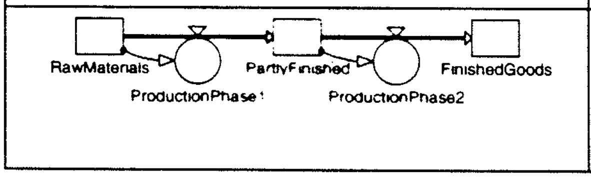
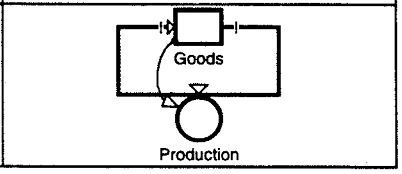
a) State transitions using scalar variables	b) State transitions using vectors
 RawMaterials PartlyFinished FinishedGoods ProductionPhase1 ProductionPhase2	

Figure 8: State transitions using scalars and vectors

If separate inflows and outflows are added, the structure in Figure 8-b may also be used to model a N-th order material delay. The DELAYMTR function is normally used to express material delays.

But if the contents of the levels implicitly defined by the DELAYMTR function need to be accessed, an explicit structure like the one in Figure 9 can be applied.

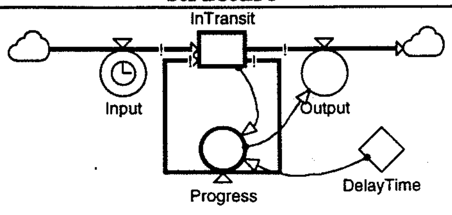
a) Diagram of material delay structure	b) Associated equations
	level InTransit(p=RPhases) = 0 +dt*(Input |p = FIRST(p);0) +dt*(Progress(p-1)|p > FIRST(p);0) -dt*(Progress(p) |p < LAST(p) ;0) -dt*(Output |p = LAST(p) ;0) aux Input = PULSE(5, 5, 0) aux Output = Progress(LAST(RPhases)) aux Progress(RPhases) = InTransit/ (DelayTime/COUNT(RPhases)) const DelayTime = 10

Figure 9: Material delay structure using vectors

State information

If a system goes through discrete states, this may sometimes be modeled using a vector with the same number of elements as there are system states. Each time the system state changes, the elements of the state vector are shifted cyclically, using the SHIFTCIF function.

As an example, think of a process that may be in one of three distinct states; *StateA*, *StateB*, and *StateC*. Also assume that a variable *NewState* will be set to true whenever the system should be switched to the next state. Furthermore, assume that state transitions are cyclic, i.e., that *StateA* follows *StateC*.

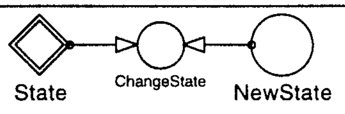
a) Diagram of state transition	b) Associated equations
	range RState = StateA, StateB, StateC aux ChangeState = SHIFTCIF(NewState, State) const State(RState) = [1,0,0] aux NewState = ... "Some boolean expression"

Figure 10: Discrete system states

Observe how the system states are defined as elements of an enumeration range. Initially, the first state of the *State* vector is true (1), and the others false (0). Whenever *NewState* is true, the r.h.s. of the *ChangeState* definition will shift the elements of *State* one position to the right, changing the state from [1, 0, 0] to [0, 1, 0] the first time, and to [0, 0, 1] the second time, and to [1, 0, 0] the third time, etc.

Testing for a given state may be done using the following expressions:

State(StateA) – will be true whenever the system is in the first state
State(StateB) – will be true whenever the system is in the second state
State(StateC) – will be true whenever the system is in the third state

Object attributes

A modeler is sometimes interested in keeping track, not only of the number of objects, but also their attributes or properties. As an example, we may be working on a production model, where products have attributes like lifetime, functionality, price, etc.

In Figure 11 the scalar and array version of the attribute part of a model is presented. In Figure 11-b a range *RAttribute* is defined like this:

range RAttribute = EPrice, EFunctionality, ELifetime, EInventory

The price of our product will be available as *Product(EPrice)*. The other attributes are accessed correspondingly.

The significance of this solution is that all attributes are moved along as a co-flow when the object (here product) flows through the system or is accessed via links. The addition of more attributes can be done without changing the structure of the model.

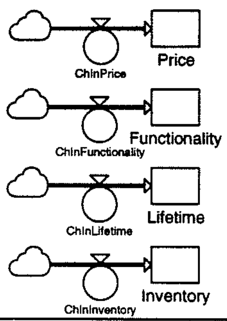
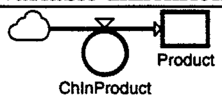
a) Separate variables for each attribute	**b) Attributes aggregated in variable dimension**
 Price ChlnPrice Functionality ChlnFunctionality Lifetime ChlnLifetime Inventory ChlnInventory	

Figure 11: Separate and aggregate attributes

History of values

As a final example of how arrays can be useful to the modeler, let us look at the problem of keeping track of previous variable values[11]. Let us assume that we construct an economic model, and that we want to keep track of the income during the recent 12 months.

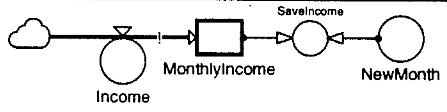
Structure for remembering values	**Associated equations**	
	level	MonthlyIncome(i=1..12) = 0 +dt*(Income WHEN i=1 BUT 0)
	aux	Income = ... "Income rate"
	aux	NewMonth = TIMECYCLE(1, 1)
	aux	SaveIncome = SHIFTLIF(NewMonth, MonthlyIncome)

Figure 12: Model for remembering values

This model accumulates *Income* in the first element of the *MonthlyIncome* vector, which has 12 elements. Every month, the elements of *MonthlyIncome* are shifted one position, and the first element is cleared (all this is done by the SHIFTLIF function).

Main differences between arrays in DYNAMO and POWERSIM

In DYNAMO, index variables and ranges are combined into FOR variables. Each dimension of a variable must be defined in a separate statement. POWERSIM's introduction of guards and constrained expressions makes it always possible to define a multi-dimensional variable in a single statement. This is important when working in a graphical environment, and also removes redundancy from the language (Powersim, p. 67). DYNAMO does not have array literals, and multi-dimensional array constants must be defined one dimension at the time. DYNAMO does not have functions returning arrays.

[11]Note also that the SAMPLEIF function may be used for remembering a value for later use. Related functions are HIVAL and LOVAL, which remember the highest and lowest value of a variable, respectively, up to the current time of the simulation.

Adding arrays to scalar models

The normal way of constructing an array model, is by first making and testing a scalar version of the model. Then a set of ranges are defined, and selected variables are given dimensions based on the defined ranges. POWERSIM has been designed to make the transition from scalar models to array structures as smooth and easy as possible. Most scalar equations will be accepted without change also after changing a variable to an array. This is achieved through generation of default flow definitions and implicit definition of array subscripts.

Experiences from applications

The array features of POWERSIM have been used with success in several large-scale projects (Powersim 1993). Without the array capability, many modeling problems are not practically solvable. Even models that are solvable using only scalars, sometimes become much easier to develop, explain and maintain using arrays. In conclusion, the family of simulation problems that are best solved using a system dynamics tool has been extended a great deal through POWERSIM's array mechanism.

References and readings

ModellData 1993. *Powersim.* ISBN 82-91403-00-7
Powersim November 1993. *The Powersim Newsletter*
Pugh-Roberts Associates, Inc. *Professional DYNAMO Plus*