The Neutral Model Format for Building Simulation

Per Sahlin Axel Bring
Bris Data AB
and
Building Sciences
Royal Institute of Technology
100 44 STOCKHOLM, SWEDEN
phone +46-8-411 32 38
e-mail: plurre@engserv.kth.se

Edward F. Sowell
Dept. of Computer Science
California State University, Fullerton
Fullerton, CA 926 34
phone: +1-714-773-32 91
e-mail: sowell@fullerton.edu

ABSTRACT

The idea of a completely general simulation environment, where a user can interconnect predefined submodels freely into a tailored system model, is today a reality. In the field of building simulation, existing environments like TRNSYS and HVACSIM+ along with several new developments, allow fully coupled models of envelope, distribution systems and controls at an arbitrary level of detail. However, the ultimate usefulness of any of these tools hinges on the existence of a comprehensive library of component models and the development cost of such a library will easily exceed that of the environment itself. In this report a Neutral Model Format (NMF) is specified. NMF models can be automatically translated into the format of a number of environments. Based on NMF, independent libraries can be established, and inter-environment model exchange is likely to increase. Since the first NMF proposal in 1989, several prototype translators have been developed, model libraries have been written, and the concept has earned acceptance among experienced users. This report repeats the modelling principles underlying NMF and presents a brief reference manual. A formal syntax definition and some model examples are presented in appendices.

Version 3.02
June 1996

This work is supported in part by the Swedish Council for Building Research under contracts 870299-8, 880509-2, 890808-8, 900215-1, 900368-5, 930212-1, and 930792-0; the Swedish National Board for Technical Development under contracts 90-02868 and 90-02927; the Nordic Construction Company; and the Development Fund of the Swedish Construction Industry under contract 1066; American Society for Heating, Refrigerating and Air-conditioning Engineers under contract RP-839.

CONTENTS

1. BACKGROUND 4

2. SCOPE 5

3. MODEL STRUCTURING PRINCIPLES 7

3.1 Equation modelling 7
3.2 Component Interconnection 9
3.3 Property Inheritance 11
3.4 Hierarchical Decomposition 12
3.5 Model Validation 12

4. NMF REFERENCE MANUAL 13

4.1 Global Declarations 15
4.2 Continuous Model Elements 16
4.2.1 Equations 17
4.2.2 Assignment modelling 18
4.2.3 Links 20
4.2.4 Variables 22
4.2.5 Parameters and Model Parameters 23
4.2.6 Parameter Processing 24
4.2.7 Functions 24
4.3 Special Functions 25
4.3.1 Error Subroutine 25
4.3.2 Event functions 25
4.3.3 Model linearization 26
4.3.4 Delays 27

5. ACKNOWLEDGEMENTS 27

6. REFERENCES 27

7. APPENDIX 1 SYNTAX DEFINITION FOR NMF 30

8. APPENDIX 2 EXAMPLES OF MODEL DEFINITIONS 39

9. APPENDIX 3 CHANGES IN VERSION 3 56

10. APPENDIX 4 A WORKED SYSTEM EXAMPLE 59

Preface to version 3.02

Version 2 of this report was published in August 1992, version 3 and 3.01 during the spring of 1994. The present version incorporates errors and minor changes that have been recorded since then. Many errors have been reported by attentive readers, but the bulk of corrections is a side effect of the work of Dr. Pavel Grozman, in the development of the ASHRAE (American Society of Heating, Refrigerating and Air-conditioning Engineers) NMF Translator.

Another deliverable of the ASHRAE NMF Project (RP-839) is a much needed NMF Handbook [SAHLIN 1996]. Up until now, there has been no pedagogical NMF material available. Due to the existence of the Handbook, some of the general NMF discussion can be removed from this text. This process has been initiated in this edition, and it is expected that future versions will have even more reference character.

The changes in the report, which have been incorporated into both the text and the formal syntax, have been marked with a solid vertical line in the right margin, for the benefit of those who are acquainted with the previous version.

NMF version 3.02 and the draft version of this report were approved by the ASHRAE NMF Committee at the Atlanta meeting, February 20, 1996.

Since the previous version was published, an NMF discussion list and ftp area has been started. To join the list, send an e-mail to

mailbase@mailbase.ac.uk

The Subject line is irrelevant, but the Body should read:

join ibpsa-nmf <yourfirstname> <yourlastname>

Your e-mail address is automatically recorded, so join from your own account. Recent NMF developments are reported to this list and a small archive of NMF related material is also available.

Examples of component models are found in Appendices 2 and 4. Those in Appendix 2 are primarily chosen to illustrate features of the format, and are sometimes referenced in the main text. Appendix 4 contains a worked system example together with the constituent models. The example was developed in cooperation with David Lorenzetti of MIT.

1. BACKGROUND

The Neutral Model Format (NMF) was first proposed at the Building Simulation '89 conference in Vancouver, Canada. The basic objective of NMF is to provide a *common format of model expression* for a number of existing and emerging *Simulation Environments*, e.g. TRNSYS, HVACSIM+, ALLAN, CLIM 2000, EKS, IDA, SPARK. All of these are similar in that user defined mathematical models of components are expressed in separate modules that the user can interconnect as needed to define the wanted system model. At a given level of idealization, a typical component model for any of these simulation environments can be expressed as a system of ordinary differential and algebraic equations. NMF draws on this similarity in order to define a standard source format from which models can be automatically or semi-automatically translated into the specific format of any environment.

A second objective is to provide a *natural form of model expression*, i.e. a form which makes it easy and fast to express new models for any environment - a form which appeals to engineers as well as to well trained simulation experts.

Given such a format a number of benefits can be expected. First, a single or several *common model libraries* can be formed, which will greatly enhance the usefulness of any individual environment. Secondly, *informal model communication* between users of different environments can be expected to increase resulting in higher model quality and usage. Finally, enhanced model and user mobility between environments will result in *repeated environment comparisons* and thereby provide more accurate information about relative environment performance.

The purpose of the Vancouver paper [SAHLIN 1989] was to present the principal idea and obtain feedback from the Building Simulation community regarding the attractiveness and feasibility of the concept as such. Since then a number of positive things have occurred.

- The format has received sustained interest from simulation environment developers as well as from experienced modellers, most of whom have found the format useful for practical modelling and have provided valuable criticism.

- Translators have been developed for the SPARK , IDA, TRNSYS, HVACSIM+, and ESACAP simulation environments proving feasibility of the concept

- Several realistic model libraries have been developed in, or translated into NMF.

- An ASHRAE (American Society of Heating, Refrigerating and Air-Conditioning Engineers) subcommittee of Technical Committee 4.7 (Energy Calculations) has been formed to act for the introduction of NMF as an ASHRAE standard for model definition.

These events have triggered a need for an updated introduction and a more complete NMF reference, hence, the present report. In Section 2, the scope of the NMF effort is more carefully discussed, leading up to an examination of the principal concepts in Section 3 and a format reference manual in Section 4. Appendix 1 contains a formal,

but human readable, BNF-oriented syntax definition and Appendix 2 holds some model examples. The changes in version 3 of NMF are listed in Appendix 3 and Appendix 4 presents a small system, its components in NMF and some results.

2. SCOPE

Although NMF has been developed primarily to cater to the needs of the building simulation field, there is no built-in restriction that limits the usage to this area. In fact, it is naturally our hope that NMF libraries will be developed in related fields. The present focus is merely due to the better likelihood of obtaining a common agreement within a single application. However, there are some other more definite boundaries to the range of applicability of NMF that we shall try to examine in the following. The remainder of this section may safely be omitted on first reading, but it is our experience that most "second generation" questions pertain to these issues.

Most important is to recognize the difference between NMF and more complete modelling languages. A complete modelling language attempts to express all there is to know about every model for (usually) a single simulation environment, i.e. the language is essentially the sole mode of communication with the designated environment. Examples of modern modelling languages are DYMOLA, LICS [ELMQVIST 1978, 1986] and OMOLA [ANDERSSON 1990]. Another example is CSSL (Continuous System Simulation Language) which was defined in 1968 and has had, and still has, a tremendous impact on the simulation field. The momentum of the CSSL language has resulted in the development of several simulation environments *for* CSSL.

NMF, on the other hand, attempts to define a standard way of expressing models for a number of given environments of similar type. This is obviously more difficult, since not only syntax but also semantic content of model definitions varies considerably between environments. In fact, if it were not for the universal language of mathematics underlying it all, the project would probably be quite unrealistic.

For these reasons, NMF is presently limited to the *primitive*, or *leaf*, models of the target environments, while no attempt is made to standardize *composite* models and other structures. To put it in perhaps more familiar terms, the present NMF will be able to generate a large class of TRNSYS *types* (primitive models) but not TRNSYS *decks* (composite models plus additional information.). Since the original NMF proposal a number of extensions have been proposed by several researchers. The most recent proposal attempts to compile input from several proposers and covers also composite or system models in NMF [SAHLIN 1995]

Generally speaking, one could say that the priorities in defining NMF have been:

1. multi environment compatibility,
2. intuitiveness,
3. completeness;

whereas the reverse ordering seems to apply to most alternative efforts of modelling language definition.

The present version of NMF is furthermore restricted to models with a *piecewise continuous* behavior, i.e. basically models that can be expressed in terms of equations (with some extensions.) This includes the absolute majority of models in use today in thermodynamical modelling, e.g. walls, windows, heat exchangers, pipes, controls etc. Some extensions of basic equations, explained further in Section 4.2.2, allow definitions of models with *hysteresis*, such as thermostats, actuators with backlash and, in principle, also models with delay. Models with delay are easy to express in this framework, but they are (in any framework) very difficult to treat numerically even with modest requirements of reliability. We will return to the question of numerical performance of target environments in a moment.

Models that are described in discrete time are currently not encompassed. Examples of such models are micro processor based controllers, in the case when the sampling frequency is an interesting simulation parameter, and also models that are expressed in discrete time for reasons of numerical efficiency, such as building simulation style transfer function models (based on the z-transform.)

The question of differing numerical strengths of target environments is important and we shall devote some attention to it here. At a first glance, a reasonable policy might be to require that every NMF model should run in every environment. However, this would be extremely restrictive and could potentially hamper numerical development, which is the last thing we would like to see. Therefore, the present policy is that an *approved* model should execute in some *generally available,* general purpose simulation environment. Naturally, this might complicate the process of transferring common library models to a local environment library, and also potentially diminish the incentive to submit advanced local models to a common library, but it is still felt that this is the only viable policy.

In the sequel it will be assumed that all models are to be subjected to a review procedure for inclusion in common model libraries, although as we have seen, NMF can be useful even without the existence of large common libraries.

Some aspects of library writing that have not yet been treated in depth in conjunction with NMF include model documentation guidelines, modelling methodology and model library architecture. We shall in the remainder of this section say something about these issues.

It is reasonable to demand that models in a common library are accompanied by a consistent, well structured text, explaining e.g. intended usage of the model, assumptions made, validations made etc. The definition of general - not too cumbersome - guidelines is not completely straightforward since they must apply to anything from a simple thermal conductance, to e.g. a comprehensive dynamical cooling coil model or to an intricate controller model. A number of efforts have been made in this direction for the field of building simulation, e.g. [CLARKE 1984, DUBOIS 1988 and RONGERE 1992].

The term modelling methodology refers to a structured procedure to follow in the process of defining an appropriate set of models for a given area of application and level of approximation. An important side effect is also a well structured documentation. The most comprehensive work, of which we are aware, in this area with application to thermodynamical systems is that of [RONGERE 1992] and co-workers at EDF in France.

NMF is based on a few model structuring principles that will be discussed in the next section. These are intended to encourage structured modelling much in the same way as some programming languages support well structured programming. However, given these principles we have found that the importance of good *model library architecture* hardly can be overestimated. The problem is similar to that of defining a proper set of base classes in object oriented programming. Perhaps it is even more demanding in our case, since models tend to be reused to a higher degree than is usually the case with object classes. This is an area where much work is needed, in the form of well designed sample libraries, rather than in general guidelines and recommendations.

3. MODEL STRUCTURING PRINCIPLES

NMF is based on a few model structuring principles. Much of this is inspired by the work of [ELMQVIST 1986] and [MATTSSON 1988].

1. Continuous models are expressed in terms of equations
2. Variables and interconnection links are typed
3. Models can inherit properties from ancestors
4. Large models must allow hierarchical decomposition
5. Validation is integrated into the modelling process

The principles are briefly described and defended below.

3.1 Equation modelling

The choice of equations as the basic vehicle for description of internal model behavior is by no means indisputable. There are alternative formalisms that with some extensions provide the necessary generality, e.g. bond graphs and electrical analogies. The in-depth discussion of the considerations that lie behind our choice of equations is beyond our purposes here. Two of the main arguments that support our choice are: (1) Equations are familiar to modellers, i.e. models are legible (in text form) without special training and (2) much of the beauty and nice numerical properties of alternative representations are lost when they are made sufficiently general.

Continuous NMF models are described by a fully implicit differential-algebraic system of equations which for the general case can be written

$$f(\boldsymbol{x}, \boldsymbol{x}', \boldsymbol{p}, t) = 0,$$

where $\boldsymbol{f}$ is a vector function of the variable vector $\boldsymbol{x}$, its time derivative $\boldsymbol{x'}$, a parameter vector $\mathbf{p}$, and time t. In all cases of interest here, this system of equations will be underdetermined; some of the $\boldsymbol{x}$:s will have to be given as functions of time.

The remainder of this subsection will be devoted to a discussion about the merits of *input-output free models*. Readers, who are already convinced in this issue, may omit the following without loss of continuity. Let us, for the sake of the discussion, separate between the *equation model* of a component and a *problem* for the same component, where the problem is the underdetermined equation model *together* with a selection of given variables. For example, the equation model of a thermal resistance may be written

$$0 = q - U\,A\,(t1 - t2),$$

where q is the heat flow through the resistance and $t1$ and $t2$ are the terminal temperatures. Now, for this simple one-equation model three different problems, i.e. combinations of given and calculated variables, may be posed:

1. $t1$ and $t2$ given and q calculated
2. $t1$ and q given and $t2$ calculated
3. $t2$ and q given and $t1$ calculated

All three problems are *well posed*, provided U and A are nonzero. In the following, well posed will be used in the sense: able to produce a locally unique non-trivial solution. For more complex models only some selections of given variables will yield well posed problems.

Each component model in most current simulation environments, e.g. TRNSYS and HVACSIM+, is described as an equation model *along with a single input-output selection* (a problem in our sense). The component modeller makes this selection when the model type routine is written.

The pre-selection of given variables leads in some cases to limitations in the actual use of the models. Frequently a system modeller, using available types, would like to connect the inputs of one component with the inputs of another and similarly for the outputs. This, of course, is impossible and one of the component models has to be rewritten, with a different input-output selection. The system modeller is forced to become a component modeller and write, debug and compile FORTRAN code.

These difficulties are overcome in some of the more recently proposed environments (e.g. SPARK and IDA) by leaving the input-output designation to the environment. This will substantially increase the versatility of each component model.

The automatic input-output designation in more recent environments is done by keeping equation models separate from input-output selections until the components are actually connected together. This separation is only possible if equations are declared explicitly, the way they are in NMF.

Since some environments can do without explicitly stated input-output designations in their component model format, one could argue that this information is redundant in NMF, which should be free of environment specific non-essential information. There are however several reasons for including *one possible input-output designation* (one problem) for each NMF component model. First, a viable input-output set is a part of the required validation procedure. That is, a component modeller has to demonstrate at least one well posed problem for a model. Secondly, if this information was to be left out, automatic translation would be impossible for input-output oriented environments.

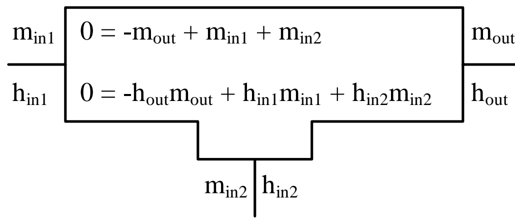
3.2 Component Interconnection

Having focused briefly on the internal behavior of component models we turn to the interconnection mechanism between them. Little attention has been devoted to this topic in many of the past discussions on the development of common component libraries, although model reuse and exchange have been the primary motivations. However, one should be aware that sets of components developed by various groups will remain to be incompatible, even when stored in a common library, unless a structured way of constructing inter-component links is imposed. Otherwise, the sockets and the prongs simply will not fit together.

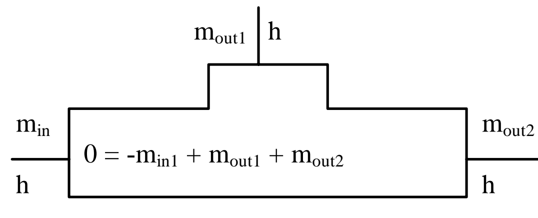
The development of a set of component models for a simulation task involves numerous decisions, some of which are crucial and others which are less fundamental in nature. Unfortunately, all of these decisions, not just the crucial ones, will later on influence the compatibility with other models. It is our aim here to provide a component format which encourages compatible choices among the trivial decisions without imposing any restrictions on the fundamental ones.

One of the initial crucial decisions to be made is the choice of a set of variables that will represent the behavior of the simulated system to an appropriate degree of accuracy. For example, in a simple HVAC circuit without cooling it might be sufficient to choose dry air mass flow rate and air enthalpy as the main variables carrying information between individual components. We are referring here to the set of variables involved in the *interaction* between components; additional variables may be used internally. Once this choice of interaction variables has been done, a *compatible family* of components can be developed. For the HVAC circuit this might involve e.g. a collector, a distributor, a heating coil and a simple zone model as shown in Figure 1.

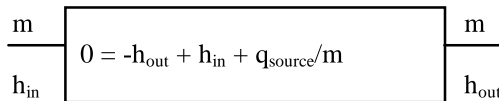
Collector



Distributor



Heating Coil



Zone

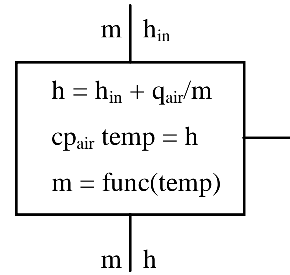


Figure 1.

The zone model has a built in control function, `func`, which determines the supply air flow rate as a function of zone temperature.

The choice of interaction variables is affected by the component model complexity. For example - if we wish to consider moist air problems - a cooling coil model should include the effect of condensation, and thus some measure of air humidity must be included in the air characterization. If the cooling coil is to be used in conjunction with the previous models, the list of interaction variables to be carried around the circuit must be expanded. The principle here is that the component in need of the most information determines the interaction set of variables. Components with smaller needs will ignore unnecessary circuit variables.

Now, let us look at some of the trivial decisions for our sample case. Although the simulation in principle can be carried out using vastly different sets of units in each component, compatibility is enhanced if common units are used. This is an area where encouragement, via access to existing models, and mild punishment, via compulsion to write additional declarations, are likely to stimulate uniformity. For the sample HVAC circuit, a similar argument can be made concerning the choice between temperature and enthalpy as an interaction variable.

A mechanism for increased compatibility in this sense is *typing of variables*. All variable types to be used in component models are declared globally. A modeller who is about to introduce a new model in the library will use already declared types whenever possible.

The next step is to declare the groups of variable types that characterize interaction within compatible families. Such a group is called a *link type*. Mass flow rate and enthalpy together and in this order is an example of such a type. Examples of typing syntax are found in Section 4.1.

The link concept also allows a user of a simulation environment to connect submodels at the interface level rather than variable by variable. This means, for example, that a fan outlet is connected with a cooling coil inlet as far as the user is concerned; in the background however several variables may be involved in the connection. Most current simulation environments, e.g. TRNSYS or HVACSIM+, operate at the variable level; the link concept would in this respect simply be ignored for these. The more important library structuring effect of link typing is still retained.

In link supporting environments, link types can be used to check whether a user is making meaningful connections. There are cases, however, when a strictly imposed typing concept is too restrictive. Controllers, for example, should be allowed to interface with various types of links. This is dealt with in NMF by providing a generic link type which can contain any number of any type of variables. An environment can then check the individual variables in the connecting links for matching types rather than the links themselves. A generic variable type is also provided in order to allow for suppressed type checking on the variable level as well. [MATTSSON 1988] discusses ways of constructing more elaborate type checking in a modelling environment.

3.3 Property Inheritance

In the discussions preceding the Vancouver NMF proposal, one issue was whether to allow inheritance of properties between models, i.e. to make NMF "object oriented." The arguments that made us settle then for a flat model class concept were all related to model simplicity, e.g.: more of "What You See Is What You Get" if one has all of a model in one place; ease of library structuring and model look up; no required user familiarity with OOP concepts etc.

However, modelling experience has led us to revise this view, and we are presently working on a single-inheritance mechanism. The basic idea is to let models of the same physical component with successively increasing level of complexity inherit properties from less complicated ancestors, i.e. another dimension is added to the compatible family genealogy. For example, a collector model with massflow-enthalpy-relative_humidity links would inherit most of its properties from a massflow-enthalpy collector.

The NMF version that is formally presented later on in this report does not encompass the (few) additional concepts that are necessary for model inheritance.

A more detailed proposal for NMF model inheritance can be found in [SAHLIN 1995].

3.4 Hierarchical Decomposition

Another fundamental concept for structured modelling is hierarchical submodel decomposition, i.e. one submodel within another in multiple levels. A composite building model could then, e.g., be composed of several submodels, each one representing a floor. A floor is in turn built up of several zone models, which are built from wall models, and so on. One major advantage of this method is that it enables incremental modelling and validation. A modeller can make sure that e.g. a wall model behaves properly before it is used as part of a zone model, which then is similarly validated and so on, incrementally approaching the building level. Another advantage is that good graphical interfaces can be constructed for a corresponding hierarchical presentation of a model, where a user first gets an overall view of the system and then can zoom down for successively increased levels of detail.

Although most component models in NMF will be used as part of composite or macro models on the environment level, the formatting of composite models themselves, i.e. interconnection templates, is not encompassed by the present version.

A detailed proposal regarding hierarchical models in NMF can also be found in [SAHLIN 1995].

3.5 Model Validation

There is no way to stop someone from using a library component in a non-intended way. The best thing one can do is to require extensive textual documentation to be provided along with the library entry, including the background of the underlying mathematical model.

The ambition of NMF is to make sure that the entered models make sense from a mathematical perspective. Unfortunately, even this is quite a task. Existence of solutions to nonlinear equations is a very difficult subject and no general and practical theory exists. A model may work well over a particular parameter and variable range and be ill posed over another. In the end, we are left with the component modeller's ability to write robust models and to document them properly, including their ranges of validity.

What is required for an approved NMF model is that a single problem - one input-output designation along with an equation model - is provided, and that its range of well posedness is specified. The well posedness range is specified in two different ways: first, in terms of explicit limits on the involved parameters and variables and, secondly, in terms of accompanying documentation. Responsibility for the existence of solutions for other possible input-output designations must be left to the targeted environments.

The most practical method for finding the range of well posedness of a model is *numerical testing*. The idea is that the modeller finds some algorithm for solving the designated problem, e.g. a direct iterative scheme or a general purpose differential-algebraic integrator such as DASSL [BRENAN 1989] or even a

simulation environment, and then, by numerical experimentation, finds the range of well posedness in parameter and variable space. As a minimum, it must be shown that a solution exists in the intended operational regime.

It is recommended that the default values of variables and parameters in the model, whenever this is meaningful, be chosen in such a way that they together satisfy the equations. If this is done, a new model user can make a quick test of the model in his or her environment: when IN variables and parameters are set to their defaults, solving for the OUT variables should give their defaults.

Formally, for a general component model (from Section 3.1), call the vector of the designated input set $\boldsymbol{u}$ and the corresponding output vector $\boldsymbol{x}$ yielding $\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{x}', \boldsymbol{u}, \boldsymbol{p})$, where dim($\boldsymbol{f}$) = dim($\boldsymbol{x}$). Then the matrix (pencil):

$$\lambda \frac{df}{dx'} - \frac{df}{dx},$$

where λ is some scalar, must be non-singular for all but a finite number of λ:s and this must, of course, be true for the entire parameter and variable working range of the component [ERIKSSON 1992].

Some solvers take advantage of information about "undesirable inverses" of individual equations. The basic idea here is that a scalar equation, e.g. $h(x,y) = 0$, may be readily inverted to yield $x = g1(y)$ where $g1$ is a well behaved function, but the inverse $y = g2(x)$ may be problematical. One possible problem is that the function $g2$ may not be well behaved numerically. E.g. $dg2/dx$ may become infinite in the range of interest, or for environments that develop the inverses symbolically, $g2$ may not be obtainable as a closed form expression, or even if obtainable it may have poor numerical properties or be unwieldy.

4. NMF REFERENCE MANUAL

In this Section the elements of continuous NMF models are described. Model examples are given in order to support the syntax presentation but the art of model library building is not discussed. A formal syntax description is included in Appendix 1. In order to provide some overview we will start with an example of a simple NMF model and then more carefully go through each element.

```
CONTINUOUS_MODEL tq_conduction

ABSTRACT "Linear conductance"


EQUATIONS

/* heat balance */
   0 = - Q + a_u * (T1 - T2)   BAD_INVERSES ();
```

```
LINKS

/* type         name         variables... */
   TQ           Terminal_1   T1, POS_IN Q ;
   TQ           Terminal_2   T2, POS_OUT Q ;


VARIABLES

/* type         name   role [def [min        max]] description */
   Temp         T1     IN    0.   ABS_ZERO   BIG   "1st temp"
   Temp         T2     IN    0.   ABS_ZERO   BIG   "2nd temp"
   HeatFlux     Q      OUT   0.   -BIG       BIG   "flow from 1 to 2"


PARAMETERS

/* type         name   role [def [min     max]]  description */
   Area         a      S_P   1.   SMALL   BIG    "cross sect area"
   HeatCondA    u      S_P   1.   SMALL   BIG    "heat trans coeff"
   HeatCond     a_u    C_P   1.   SMALL   BIG    "a * u"


PARAMETER_PROCESSING

   a_u := a * u;


END_MODEL
```

The heart of an NMF model is the `EQUATIONS` Section, where the governing differential-algebraic equations are stated. In the example there is only one algebraic equation and no ordinary differential equations, but in the general case there may be several of each kind. Equations define relations between variables, time derivatives of regular variables, parameters, and constants. All the standard F77 floating point functions may be used and equations may also refer to user defined functions.

Any entity that may vary with time is called a variable. Variables in a model are declared separately and must be of a globally, i.e. library common, variable type. For example, the heat, `Q`, through the conductor is of the global type `HeatFlux`.

Parameters, on the other hand, remain constant throughout a simulation. They are declared and globally typed similarly to variables. Variables and parameters use the same global types called quantity types. Some parameters, `a` and `u` in the example, do not necessarily appear in any equation. They are called easy access parameters and are the ones that are actually specified by a user. A parameter processing section is executed prior to simulation, in order to calculate remaining parameters, e.g., `a_u` in the example.

Constants are common to all models and their values are permanently given by a global declaration. In the example we find the constant `ABS_ZERO` and the machine dependent constants `BIG` and `SMALL`.

Links define how a model may be connected with neighboring models. They usually correspond to ports of a physical component, e.g. the terminals of a resistor or the inlet and outlet of a fan. All variables that connect the model with neighboring

models must appear in the link declarations. There is no such things as global variables, i.e. variables that are accessible to more than one model. Links are typed globally in terms of the number and types of variables that appear in the link declaration.

4.1 Global Declarations

As previously motivated, variable (quantity) types and groups of such types, link types, are declared globally. The global declarations are then referenced from each component model declaration. Constants are also declared globally within a library of component models. Parameters share the quantity type declarations with variables.

The type declaration section is placed at the head of an NMF file. An example of some useful types are:

```
QUANTITY_TYPES

/* type name      unit              kind */

   Area           "m2"              CROSS
   Control        "dimless"         CROSS
   Density        "kg/m3"           CROSS
   Enthalpy       "J/kg"            CROSS
   Factor         "dimless"         CROSS
   HeatCap        "J/(K)"           CROSS
   HeatCapM       "J/(kg K)"        CROSS
   HeatCond       "W/(K)"           THRU
   HeatCondL      "W/(m K)"         THRU
   HeatCondA      "W/(m2 K)"        THRU
   HeatFlux       "W"               THRU
   HeatFlux_k     "kW"              THRU
   HumRatio       "kg/kg"           CROSS
   Length         "m"               CROSS
   Mass           "kg"              CROSS
   MassFlow       "kg/s"            THRU
   Pressure       "Pa"              CROSS
   RadiationA     "W/m2"            THRU
   Temp           "Deg-C"           CROSS
   Volume         "m3"              CROSS
   VolFlow        "m3/s"            THRU
   VolFlow_h      "m3/h"  THRU

/* CROSS = Potential, non-directional */
/* THRU  = Flow, directional */

LINK_TYPES

/* type name        variable types... */

   Q                (HeatFlux)
   T                (Temp)
   TQ               (Temp, HeatFlux)

   M                (MassFlow)
   MT               (MassFlow, Temp)

   PM               (Pressure, MassFlow)
```

```
   PMT               (Pressure, MassFlow, Temp)

   MoistAir          (Pressure, MassFlow, Temp, HumRatio)
   BidirFlow         (Pressure, MassFlow, Enthalpy, HeatFlux)
   HeatSun           (Temp, HeatFlux, RadiationA, RadiationA)

   ControlLink       (Control)
   ControlLimit      (Control, Control)

CONSTANTS

/* name           value      unit */

   ABS_ZERO       -273.16    "Deg-C"         /* absolute zero temp */
   BOLTZ          5.67E-8    "W/(m2 K4)"     /* Stefan Boltz const */
   CP_AIR         1006.      "J/(kg K)"      /* air specific heat */
   CP_VAP         1805.      "J/(kg K)"      /* watervapor spec heat */
   CP_WAT         4187.      "J/(kg K)"      /* water specific heat */
   CV_AIR         720.       "J/(kg K)"      /* air specific heat */
   G              9.81       "m/s2"          /* gravity acc */
   GASCON         287.       ""              /* general gas const */
   HF_VAP         2.501E6    "J/kg"          /* water vaporizat heat */
   LAMBDA_AIR     0.0243     "W/(m K)"       /* air thermal conduc */
   LAMBDA_WAT     0.554      "W/(m K)"       /* water thermal conduc */
   P_ATM_0        1.013E5    "Pa"            /* standard air press */
   PI             3.1415927  "dimless"       /* the pi number */
   PRANDTL_AIR    0.71       "dimless"       /* air Prantl number */
   RHO_AIR        1.2        "kg/m3"         /* air density */
   RHO_WAT        1000.      "kg/m3"         /* water density */
   VISC_WAT       1.E-3      "kg/(m s)"      /* water dynamic visc */
```

The first two fields of a quantity type declaration need no explanation, but "`kind`" may not be familiar. (`kind` is not relevant when quantity types are used for parameters.) All variables can be categorized as being of, either direction-dependent flow type (e.g. mass flow, heat flow, electrical current, torque and force), or direction-independent potential type (e.g. temperature, pressure, enthalpy, voltage and position). The physics of flow-type variables says that they should sum to zero when two such variables are connected together. They are traditionally called `THRU` variables and will be called so here as well. Potential-type variables, on the other hand, are set equal to each other when connected. They are called `CROSS` variables.

Functions and Subroutines can either be declared globally, for access in several models, or locally within a model, for exclusive use within that model. Global and local function declarations are syntactically identical. This is discussed further in Sections 4.2.6-7.

4.2 Continuous Model Elements

The elements of continuous models will be introduced incrementally. The reader is advised to study a collection of NMF models from a familiar field of application in parallel. Some models for thermal modelling of buildings are collected in Appendix 2 for reference.

4.2.1 Equations

As previously motivated, the internal behavior of continuous models is described by a system of scalar equations, each of which is written

<expression> = <expression>,

where an expression may be a single variable, a first order time derivative, a parameter, a number, or some mathematical combination of these. The aim is to keep the syntax as "natural" as possible. Expressions may also include references to separately defined functions written in a regular programming language.

One or more ordinary differential and/or algebraic equations may be stated and their individual order is of no formal consequence to the solution procedure. It is important to understand that NMF only states a mathematical model. It does not suggest any procedure for solving the equations. This is handled automatically by the target environment in some cases (e.g. IDA and SPARK) or by a combination of environment and translator software in others (e.g. TRNSYS and HVACSIM+.) An example of a legal model is

```
EQUATIONS

  X1' = f1(X1,X2,X3);
  0 = g1(X1,X2,X3);
  f2(X1,X2,X3) = X3*X2';
```

where `X`:s are regular variables, `X'`:s time derivatives, and `f`:s and `g`:s user defined functions.

Case has no formal meaning in NMF, but can be used to enhance legibility of model definitions. Conventionally, capital letters are used for keywords and constants, while parameters are written in lower case letters, and variables in lower case but with the first letter in uppercase. Time derivatives are marked by an apostrophe immediately following the variable name, before indices for indexed variables.

Piecewise defined equations are handled by the *conditional expression*, for example:

```
/* required supply air mass flow rate */
   M = IF Temp >= tmax THEN
          mmin
       ELSE_IF Temp <= tmin THEN
          mmax
       ELSE
          ((Temp - tmin)*mmin + (tmax - Temp)*mmax)/(tmax - tmin)
       END_IF   BAD_INVERSES (Temp);
```

Dependencies like these can of course be hidden in external functions. It does however often make the model more legible if they are declared explicitly. The optional list of `BAD_INVERSES` has been tagged on to the sample equation in order to signal the danger of trying to solve for `Temp`. One can chose to instead specify `GOOD_INVERSES`, if this is more convenient.

The number of equations in a continuous model may also be flexible to a certain extent, thus allowing e.g. simple finite difference models. The special constructs for indexed variables are, FOR for stating several scalar equations, and SUM for sums over indexed expressions, for example:

```
FOR i = 2, (n-1)
  p*T'[i] = T[i-1] - 2*T[i] + T[i+1] + SUM j = 1, m  Q[i,j] END_SUM;
END_FOR
```

where i and j are counter variables which are not declared separately, and n and m are model parameters. These form a special class of parameters for defining index ranges; they are further explained in Section 4.2.5. The complete syntax of FOR and SUM is located in Appendix 1. There are also some examples of usage in Appendix 2.

The variables Time and Timestep, for global time and integration timestep in seconds, are an exception to the principle of no global variables. They are available in each model without separate declaration.

Time in NMF is always measured in seconds. Equations are then automatically converted to suit the time unit of the target environment.

4.2.2 Assignment modelling

One purpose of NMF is to bring forth the advantages of equation based modelling in contrast to traditional assignment modelling with fixed input output structure. This has been discussed at some length in Section 3.1. However, there are some cases when it is convenient and even necessary to combine assignments with equations.

One instance of convenience is when a set of equations contain the same expression in several places, e.g.

```
g1(f1(X1, X2, X3)) = X1;
g2(f1(X1, X2, X3)) = X2;
```

If the the common expression is costly to evaluate, one would prefer to evaluate it only once and store the result in a help variable,

```
Help := f1(X1, X2, X3);
g1(Help) = X1;
g2(Help) = X2;
```

One might argue that cases like these could be spotted by a clever translator and the help variable could be introduced automatically in the translation process. This is certainly within reach of compiler technology of today, but we have, nevertheless, chosen to put the burden on the modeller rather than the translator. This is done in order to make translator writing relatively straightforward, knowing that the emergence of translators within reasonable time is crucial to any standardization attempt.

Help variables (declared `LOC`) are also used to carry information from subroutine calls to equation evaluation, e.g.

```
CALL sub1 (X1, X2, Help1, Help2);
g1(Help1) = X1;
g2(Help2) = X2;
```

A case when pure equation modelling is insufficient - and assignments are necessary - is when a model has several distinct states or modes, i.e. the model must remember from one evaluation to the next what state it is in. The typical example is a thermostat, which must remember its previous output state in the dead band,

```
State := IF T > t_high THEN 0
         ELSE_IF T < t_low THEN 1
         ELSE State
         END_IF;
```

where `T` is a variable in the usual sense, `t_high` and `t_low` are parameters. `State` is an *assigned state* which keeps its value from one timestep to the next. The formal difference between an assigned state and a pure help variable is that a help variable is assigned to before it is referred to, while the opposite is true for an assigned state.

Assigned states (declared `A_S`) translate directly to variables that are put in the S common area in TRNSYS or SAVED array in HVACSIM+. The updating of stored variables is complicated for users of these environments, since only the value of the last iteration of a timestep is to be retained. An NMF translator would introduce the updating code automatically in a translated model for TRNSYS and HVACSIM+. In IDA, assigned states are updated locally by the component model in each iteration, but reset again globally prior to the next iteration, to the value they received at the last iteration of the previous timestep.

Modelling with assigned states is usually associated with discontinuous models, like a thermostat. The discussion about numerical treatment of these models leads quite far and it has been treated in a separate work [ERIKSSON 1992]. Here, we will be concentrating on the syntax and semantics of model expression only.

Assignments may be freely mixed with equations. An *assigned state* must be assigned to once, it may not be assigned to repeatedly. *Help variables* may or may not be assigned to during the execution of a model, but they may not be assigned to repeatedly. Help variables are of course only meaningful if assigned to before the equations where they are referred to.

No effort has been made to provide a complete set of programming constructs for assigned variables. It is important to recognize that they are merely intended to complement equations and not to replace them. Aside from the assignment statement itself, an `IF-THEN-ELSE` construct is provided and the previously discussed `FOR` statement may also be used. Some examples will illustrate the rules.

Throughout the examples, `State`, `Help` and `Help1` are assigned variables; `X`, `X1` and `Tot` are regular variables:

```
IF <condition> THEN
   Help := 2;
   Help1 := 1;
ELSE_IF <condition> THEN
   Help := 3;
   Help1 := 4;
END_IF

<equations>
```

- OK

....

```
FOR i = 1, n
   State[i] := X[i]*X1[i];
END_FOR
```

- OK

....

```
FOR i = 1, n

   Help := f(i, X);
   X'[i] = Help;
   X1'[i] = Help;

END_FOR
```

- NOT PERMITTED, Help is assigned repeatedly

....

```
FOR i = 1, n

   Help1[i] := f(i, X);
   X'[i] = Help1[i];
   X1'[i] = Help1[i];

END_FOR

0 = Tot - SUM j = 1,n Help1[j] END_SUM
```

- OK

4.2.3 Links

All variables that connect the model with neighboring models must appear in a link declaration. The same variable may appear in more than one link. Links may also contain variables that do not appear in any equation, e.g. pressure in a node where no pressure drops are calculated (cf. example 7 in Appendix 2.)

A link must be, either of a globally declared link type, or `GENERIC`. Links that are declared `GENERIC` may contain any number of variables of any type. Such a link may be connected with a neighboring link of any type with the same number of slots and where each connected pair of variables have matching types, i.e. types are identical or one of them is `GENERIC`.

Each `THRU` variable in the link is specified in terms of its direction of definition, i.e. the variable name is preceded by either `POS_IN` or `POS_OUT`.

Link variables are, in the present NMF version, limited to scalars. This restriction is likely to be relaxed if, in the future, NMF is extended further in the direction of partial differential equations.

Assigned variables may not appear in links of continuous models. This restriction is made in order to prevent unintended use of assigned variables.

Links may be indexed, thus allowing a flexible number of ports to a model (cf. example 1 in Appendix 2.) Variables in an indexed link may be vector but not matrix elements. The element index must be the counter variable alone, not some function of it, nor a constant number or a parameter. The limits in the `FOR` loop must be 1 and the same model parameter that dimensions the variable vectors in the link. This restriction might be relaxed in the future, hence the present link statement construction.

```
  LINKS

/* type       name         variables    */
   FOR  i = 1, n
     PMT      port[i]      Pressure, POS_IN MassFlow[i], Temp[i]
   END_FOR
```

- OK, if n dimensions `Massflow` and `Temp` in their declarations

```
........................

   FOR  i = 1, n
     PMT      port[i]      Pressure, POS_IN MassFlow[i], Temp[i,i]
   END_FOR
```

- NOT PERMITTED, `Temp` is a matrix

```
........................

   FOR  i = 1, n
     PMT      inlet[i]     Pressure, POS_IN MassFlow[i], Temp[1]
   END_FOR
```

- NOT PERMITTED, `Temp`'s index is a number

```
.........................

   FOR  i = 1, n
```

```
    PMT       outlet[i]   Pressure, POS_IN MassFlow[i], Temp[n]
  END_FOR
```

- NOT PERMITTED, `Temp`'s index is a parameter

4.2.4 Variables

Variables are separated in two main groups, regular and assigned, with each group further subdivided in two. Regular variables receive their values in the global equation solving procedure. Assigned variables receive theirs locally within a model by assignment. Assigned variables can be either help variables (`LOC`) or assigned states (`A_S`). The difference between the two has been discussed in Section 4.2.2. Regular variables are separated into given (`IN`) and calculated (`OUT`). As discussed in Sections 3.1 and 3.4, the modeller is required to specify one well posed problem for each model, i.e. one of all possible partitions into given and calculated variables. The number of `OUT` variables must always be equal to the number of equations in a model.

Each continuous model variable is declared in seven aspects:

1. Type. Each type that is referred to must be either globally declared or GENERIC. GENERIC variables that appear in links are treated the same way as CROSS variables, i.e. they are connected without any sign change to account for direction.

2. Identifier. For vector or matrix variables upper limits of index ranges are given in terms of model parameters within square brackets. The lower index is 1. hereis

3. Role. As mentioned earlier variables are cast to play one of four roles: IN, OUT, LOC, or A_S.

4. Default value. Most environments will provide defaults for initial values (of dynamic variables and assigned states) and for initial value guesses (for algebraic variables). The default value is redundant for LOC variables. It may be omitted for all variables. If left out Default defaults to 0. Globally declared constants may be used for default values and for min and max.

5 & 6. Minimum and maximum limits of the allowed range. Each variable is optionally given a range, within which the model is valid. If left out, they default to the global constants -BIG and BIG. SMALL and BIG are global machine dependent constants for small and big floating point numbers.

7. Description. Explanatory text string within quotation marks, no more than 80 characters excluding carriage returns, tabs and additional blanks, all of which may be filtered away by the target environment.

Variables that only appear in the links (interfaces) of a model - i.e. which do not appear in any of the equations - are also declared. For these variables, role should be set to `IN` (cf. example 7 in Appendix 2). Examples of variable declarations follow:

```
VARIABLES

/* type     name         role  [def  [min       max]]  description */

GENERIC     Bf           A_S    0.1   SMALL     .999  "Coil bypass factor
                                                       according to Carrier
                                                       (dimless)"
GENERIC     C_r_r        LOC    0.28  SMALL     BIG   "Capacity rate ratio
                                                       (dimless)"
HumRatio    G_i          IN     0.2   0         1     "Inlet air humidity
                                                       ratio"
Pressure    P_o          OUT    10                    "Outlet pressure"
Temp        T_ao         OUT    10    ABS_ZERO  100   "Outlet air dry bulb
                                                       temperature"
HeatFlux    Q[n_srf]     OUT                          "Heat flowing into
                                                       each surface"
Temp        T[n_x, n_y]IN       20    ABS_ZERO  30    "Slab temp field"
RadiationA  R[n_srf,n_srf]
                         LOC                          "Surface radiation
                                                       exchange"
```

4.2.5 Parameters and Model Parameters

Parameters are used to adapt a generally formulated equation model to the behavior of an actual device. They are declared under two separate headings: `MODEL_PARAMETERS` and `PARAMETERS`. The former allow a user to adapt a model structurally. They are integers that dimension arrays and matrices but may also be used as regular parameters.

Parameters use the same global quantity types as variables. Their declaration is identical to a variable declaration, except for the Role field which takes other values. For `PARAMETERS` the possible roles are `S_P` and `C_P`, for `MODEL_PARAMETERS` they are `SMP` and `CMP`. `S_P` and `SMP` are used for parameters specified by the user, while `C_P` and `CMP` are reserved for parameters calculated within the model (see next section). `def`, `min` and `max` may be left out in the same way as for variables (either all three or `min` and `max`). For `PARAMETERS` they default to `0`, `-BIG` and `BIG`, while for `MODEL_PARAMETERS` they are set to `1`, `1` and `BIGINT` if left out. `PARAMETERS` may be vectors or matrices but `MODEL_PARAMETERS` are always scalar integers. Examples:

```
MODEL_PARAMETERS

/* type     name         role [def [min  max]]   description */

   INT      n_srf        CMP    4    2    BIGINT  "Number of surfaces"

   INT      n_cir        SMP    3    0    BIGINT  "Number of parallel
                                                   water circuits"


PARAMETERS

/* type         name        role [def [min   max]] description */
```

```
   Length       d_o_tube   S_P    .02  SMALL BIG   "Tube outside
                                                    diameter"
   GENERIC      fl_res     C_P    .3   SMALL BIG   "Flow resistance param
                                                    on air side(Pa*s2/m6)"
   HeatFlux_k   q_s[n_srf] S_P                     "Heat sources or sinks
                                                    at each surface"
```

4.2.6 Parameter Processing

Except for globally defined constants, all the various named coefficients of the equation model are declared as parameters, but in addition to this, extra parameters may be declared in a model. Frequently, the mathematical characterization of a model, i.e. the parameters that appear in the equation model are quite different from those that an engineer spontaneously would choose to specify the corresponding physical device. For example, a zone model, which accounts for long wave radiation between surfaces, would have view factors (in some form) appearing in the equation model. However, a user of such a model would normally not prefer to specify these directly but rather the sizes, orientations, and so on of the surfaces themselves.

The mapping of user specified parameters or, more informally, *easy access parameters* onto parameters used in equation models is done by an algorithm which is stated under the heading `PARAMETER_PROCESSING`. The statement repertoire and other rules are the same as for assigned variables in the `EQUATION` section, e.g. each identifier may only be assigned to once. Model parameters may also be computed, but only as monotonically increasing functions of other model parameters. Parameters and model parameters that receive values in this section must always be declared with role `C_P` and `CMP` respectively.

```
PARAMETER_PROCESSING

n_surf := n_wind + n_wall;

CALL  par_proc(n_surf, surf_height, surf_width, surf_x, surf_y, surf_z,
               surf_slope, surf_orient, surf_area, rad_coeff);
```

4.2.7 Functions

Separate functions or subroutines can be called in the `EQUATIONS` and `PARAMETER_PROCESSING` sections. They may be written in FORTRAN 77 or C, or alternatively defined with NMF statements only.

Functions may be declared either globally or locally within a model. Globally declared functions can be called from any model in the library. Global and local function declarations are identical, e.g.

```
FUNCTION

/* routine for parameter processing  */

   VOID par_proc(n, h, w, x, y, z, slope, orient, area, r_coeff)
```

```
LANGUAGE  F77

INPUT
      INT n;
      FLOAT h[n], w[n], x[n], y[n], z[n], slope[n], orient[n];

OUTPUT
      FLOAT area[n], r_coeff[n];

CODE
      SUBROUTINE PAR$P(...
C     The Fortran ...
                    .
                    .
                    .
      END
END_CODE

END_FUNCTION
```

Most modern implementations of high level programming languages allow calls to precompiled routines from other languages, usually Fortran 77, C and Pascal. An NMF translator will, when a `FUNCTION` declaration is encountered, set up the necessary declarations for a foreign function call in the target environment and invoke the proper compiler.

4.3 Special Functions

A number of so called *special functions* for interaction between models and the simulation environment are defined as part of NMF. Some of these functions will be implemented differently in different environments, e.g. due to varying numerical strategies, and they may even in some cases be dummies.

4.3.1 Error Subroutine

A global subroutine `nmf_error` can be called from a model to signal an error, print selected variables, and halt execution, e.g.

```
CALL nmf_error("error message", variable_x, variable_y);
```

The NMF translator will check the variable types and set up a proper call in the local environment. A suitable level of information could for instance be to identify each variable by component instance, component type and variable name. Expressions of variables are not permitted as arguments.

4.3.2 Event functions

Implicit solvers with variable timestep can, if the solution changes predictably, take very long steps. If a discontinuity is encountered during a timestep, special action

must be taken to signal the event to the solver, which then can use the signal to locate the discontinuity in time and select a proper timestep. Since most modern DAE solvers belong to this category, it is appropriate for the NMF to encompass such a signaling system, although, strictly speaking, the mechanism is not part of the mathematics of a model.

For solvers that lack the ability to locate events in time, dummy versions of the event functions will be provided in the form of global NMF functions.

Three different functions are involved: `event`, `eventn`, and `eventp`, which trigger events when an input signal passes through zero from - respectively - both directions, positive to negative, and negative to positive.

```
eventx(event_var, event_expr)
```

`event_expr` provides the monitored signal and `event_var` is a (declared) assigned state variable which provides storage of the value of event_expr at the last iteration of the previous timestep. The event functions do three things

1. if appropriate, signal zero crossing to the solver
2. `set event_var := event_expr`
3. return the value of `event_expr` as function value.

A thermostat with event calls can e.g. be formulated (from example 3 of Appendix 2)

```
EQUATIONS

    Out_signal = Old_signal
    BAD_INVERSES () ;

    Top := eventp(Hi, T - tmax) ;
    Bottom := eventn(Lo, T - tmin) ;
    Old_signal := IF Top > 0 THEN
                     0
                  ELSE_IF Bottom < 0 THEN
                     1
                  ELSE
                     Old_signal
                  END_IF ;
```

Event calls should always be located in assignments of assigned states. This enables the solver to "see" continuous equations at all times, since assigned states retain their values from the last iteration of the previous timestep and oscillations from one state to another is avoided during iterations. If an event has been signaled at the last iteration of a tentative timestep, a solver can discard the timestep and try to locate the event in time.

4.3.3 Model linearization

Linearization of nonlinear models can be useful means to support initial value calculation, quite independent of application field. There is reason to make this linearization an explicit feature of the NMF models. The suggested solution uses a

Boolean system function 'LINEARIZE' which can be called in the model equations to select between linear and nonlinear model alternatives. Translators for various simulation environments may implement different interpretations, the simplest being to ignore the construction by implementing a dummy LINEARIZE function. A reasonable strategy, which has been used in IDA Solver, is to plan for one or more preparatory stages at the beginning of each initial value calculation, Typically, there is only one extra *stage*, during which some components may choose to linearize their models. The stage information is requested by the component via a call LINEARIZE(n), where n is an integer constant. The detailed definition of the function is:

LINEARIZE is true if the stage number of the solver is less than or equal to n.

The solver will let the stage number vary 1,2,... until no call of LINEARIZE gets answer true.

An example, extracted from a model for a bidirectional leak:

```
/* power law mass flow equation */
   M = IF LINEARIZE(1) THEN c * Dp
       ELSE_IF Dp > 0 THEN c * Dp**k
       ELSE -c * (-Dp)**k
       END_IF
```

4.3.4 Delays

Models with delay are often used in practice, but they frequently cause numerical difficulties and very little theory on the subject exists. Nevertheless, due to the great practical advantage, a delay function should be provided in the NMF framework and the formulation and testing of such a function is planned.

5. ACKNOWLEDGEMENTS

The basic ideas behind NMF came forth during a series of discussions at the Swedish Institute of Applied Mathematics and were first presented by Ed Sowell at the Building Simulation '89 conference in Vancouver, Canada [SAHLIN 1989]. Lars Eriksson, Jon Dranger, Pavel Grozman, Harald Hermansson, Magnus Lindgren, Kjell Kolsaker, Francis Lorenz, Jean-Michel Nataf, Roger Pelletret, and many others have participated in those, and in ensuing, discussions. A special thanks also to Wille Nordqvist for the tedious work of compiling error reports and various opinions into the present version of the report.

6. REFERENCES

ANDERSSON 1990 M. Andersson "An Object-Oriented Language for Model Representation," Licentiate Thesis, Dept. of Automatic Control, LIT , Box 118, 221 00 Lund, Sweden, May 1990, CODEN:LUTFD2/(TFRT-3208)/1-102/(1990)

BRING 1993 A. Bring, P. Sahlin "Modelling Air Flows and Buildings with NMF and IDA," Conference proc. Building Simulation'93, IBPSA, Adelaide, Australia, Aug. 1993.

BRENAN 1989 K.E. Brenan, S.L. Campbell, and L.R. Petzold "Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations," North-Holland, 1989

CLARKE 1984 J.A. Clarke, L. Laret "Explanation of the Data Processor Proforma," ABACUS, Strathclyde, and Laboratoire de Physique du Batiment, Liege, working document, Dec., 1984

DUBOIS 1988 A.M. Dubois "MODEL-BASED COMPUTER AIDED MODELLING: the new perspectives for building energy simulation," communication from CSTB, B.P. 21, 06561 VALBONNE Cedex, France

ELMQUIST 1978 Elmquist, H. "A Structured Model Language for Large Continuous Systems", Phd thesis, Lund Institute of Technology

ELMQVIST 1986 H. Elmqvist "LICS: Language for Implementation of Control Systems," Dept. of Automatic Control, LIT , Box 118, 221 00 Lund, Sweden

ERIKSSON 1992 L.O. Eriksson, A. Bring, G. Söderlind "Numerical Methods for the Simulation of modular Dynamical Systems," Research Report 1992:2, Swedish Institute of Applied Mathematics, Chalmers Teknikpark, 412 88 Gothenburg, Feb. 1992

KOLSAKER 1990 K. Kolsaker "Dynamisk Simulering med IDA - Et praktisk verktöy for bygningssimulering med modellbibliotek for fjernvarmeinstallasjoner, del 1 och 2" SINTEF Varmeteknikk,7034 Trondheim, Norway, July 90, ISBN 82-585-6091-7

KOLSAKER 1991 K. Kolsaker "An NMF-Based Component Library for Fire Simulation," Conference proc. Building Simulation'91, IBPSA, Nice, France, Aug. 1991

KOLSAKER 1993 K. Kolsaker "Recent Progress in Fire Simulation using NMF and Automatic Translation to IDA," Conference proc. Building Simulation'93, IBPSA, Adelaide, Australia, Aug. 1993

LEBRUN 1988 J. Lebrun, G. Liebecq "IEA Annex 10 System Simulation - Synthesis Report," University of Liege, Oct 1988, AN10 881020-RF

MATTSSON 1988 S.E. Mattsson "On Model Structuring Concepts," Presented at 4th IFAC Symposium on Computer Aided Design in Control Systems (CADCS), Beijing, China

NATAF 1990 J.-M. Nataf, E.F. Sowell "Radiant Transfer Due to Lighting: An Example of Symbolic Model Generation for the Simulation Problem Analysis Kernel," Proc. Modeling and Simulation on Microcomputers, Society for Computer Simulation, San Diego, CA, Jan. 1990

RONGERE 1992 F.-X. Rongere, W. Ranval "A Modelling Method for Systems in Building Energy Simulation: MEMPHIS," Electricite de France, April 1992, HE 12 W 3340

SAHLIN 1989 P. Sahlin, E. Sowell "A Neutral Format for Building Simulation Models," Conference proc. Building Simulation'89, IBPSA, Vancouver, Can, June 1989

SAHLIN 1991 P. Sahlin, A. Bring "IDA Solver - a Tool for Building and Energy Systems Simulation," Conference proc. Building Simulation'91, IBPSA, Nice, France, Aug. 1991

SAHLIN 1991b P. Sahlin "IDA - a Modelling and Simulation Environment for Building Applications," Research Report 1991:2, Swedish Institute of Applied Mathematics, Chalmers Teknikpark, 412 88 Gothenburg, Dec. 1991

SAHLIN 1995 P. Sahlin, A. Bring, K. Kolsaker “Future Trends of the Neutral Model Format," Conference proc. Building Simulation'95, IBPSA, Madison, WI, USA, Aug. 1995 (available at ftp://urd.ce.kth.se/pub/reports/knthbs95.rtf)

7. APPENDIX 1 SYNTAX DEFINITION FOR NMF

Version 3.02 draft, November 1995

1 Syntax notation:

<...>	Syntagm
'x..'	Represents literal x..
:=	Define operator
[...]	Optional construct
{...}	Repeat one or more times
{...$...}	Repeat to $ or exit, {a$b} is equivalent to a[{ba}]
(...|...)	Alternative definitions
<apostrophe>	denotes a literal apostrophe.
<newline>	denotes a line break.
<tab>	denotes a tabulation.

2 Syntax

```
<specification> :=
                <foundation> { ( <model_decl> | <function_decl> ) }
```

2.1 Global declarations

```
<foundation> :=
                'QUANTITY_TYPES'        { <quantity_type_decl> }
                'LINK_TYPES'            { <link_type_decl> }
                ['CONSTANTS'            { <const_decl> } ]
<quantity_type_decl> :=    <quantity_type_name> <unit> <kind>
<link_type_decl> :=        <link_type_name> '(' { <quantity_type_spec> $ ',' } ')'
<quantity_type_spec> :=    ( <quantity_type_name> | 'GENERIC' )
<const_decl> :=            <const_name> <value> <unit>
<unit> :=                  <string>
<kind> :=                  ( 'CROSS' | 'THRU' )
```

2.2 Component models

```
<model_decl> :=            ( <continuous_decl> | <algorithmic_decl> )
<algorithmic_decl> :=      To be defined
<continuous_decl> :=
                'CONTINUOUS_MODEL'      <model_name>
                'ABSTRACT'              <string>
                'EQUATIONS'             <statements>
                'LINKS'                 <link_declarations>
                'VARIABLES'             { <var_decl> }
                [ 'MODEL_PARAMETERS'    { <model_par_decl> } ]
                [ 'PARAMETERS'          { <par_decl> } ]
                [ 'PARAMETER_PROCESSING'    <statements> ]
                [ { <function_decl> } ]
                'END_MODEL'             [ <model_name> ]
```

2.3 Equations and assignments

```
<statements> :=
                { <statement> $ ';' } [';']
<statement> :=
                (       'FOR' <count_spec> <statements> 'END_FOR'
                |       'IF' <condition> THEN [ <statements> ]
                                [ { 'ELSE_IF' <condition> 'THEN' [ <statements> ] } ]
                                [ 'ELSE' [ <statements> ] ]
                        'END_IF'
                |       <simple_statement>
                )
<simple_statement> :=           ( <equation> | <assignment> )
```

2.3.1 Equations

```
<equation> :=           <expression> '=' <expression> [ <inverse_decl> ]
<inverse_decl> :=
                (       'GOOD_INVERSES' '(' { <var_name> $ ',' }   ')'
                |       'BAD_INVERSES'  '(' [ { <var_name> $ ',' } ] ')'
                )
```

2.3.2 Assignments

```
<assignment> :=
                (       <single_id_expression> ':=' <expression>
                |       <procedure_call>
                )
<single_id_expression> :=
                (       <var_name> [ <subscripts> ]
                |       <par_name> [ <subscripts> ]
                |       <model_par_name>
                )
<procedure_call> :=
                'CALL' <function_name> '(' [ { (<string>|<expression>) $ ',' } ] ')'
```

2.4 Expressions

```
<expression> :=         [ <sign> ] <term> [ { <sign> <term> } ]
<term> :=               <factor> [ { ( '*' | '/' ) <factor> } ]
<factor> :=             <simple_expression> [ '**' <simple_expression> ]
<simple_expression> :=
                (       <unsigned_constant>
                |       <variable_expression>
                |       <parameter_expression>
                |       <function_expression>
                |       <sum_expression>
                |       <cond_expression>
                |       '(' <expression> ')'
                )
<variable_expression> :=
                        <var_name> [ <derivative> ] [ <subscripts> ]
```

```
<derivative> :=          <apostrophe>
<subscripts> :=          '[' { <subscript_expression> $ ',' } ']'
<parameter_expression> :=
                (        <par_name> [ <subscripts> ]
                |        <counter_name>
                |        <model_par_name>
                )
<function_expression> :=
                         <function_name> '(' [ { (<string> | <expression> ) $ ',' } ] ')'
<sum_expression> :=  'SUM' <count_spec> <expression> 'END_SUM'
<count_spec> :=
                         <counter_name> '='
                         <subscr_simple_expr> ',' <subscr_simple_expr>
```

2.4.1 Subscript expressions

```
<subscript_expression> :=
                [ <sign> ] <subscr_term> [ { <sign> <subscr_term> } ]
<subscr_term> :=
                <subscr_factor> [ { ( '*' | '/' ) <subscr_factor> } ]
<subscr_factor> :=
                <subscr_simple_expr> [ '**' <subscr_simple_expr> ]
<subscr_simple_expr> :=
                (        <unsigned_constant>
                |        <parameter_expression>
                |        '(' <subscript_expression> ')'
                )
```

2.4.2 Conditional expressions

```
<cond_expression> :=
                'IF' <condition> 'THEN' <expression>
                         [ { 'ELSE_IF' <condition> 'THEN' <expression> } ]
                         'ELSE' <expression>
                'END_IF'
<condition> :=           <boolean_term> [ { 'OR' <boolean_term> } ]
<boolean_term> :=
                         <boolean_factor> [ { 'AND' <boolean_factor> } ]
<boolean_factor> :=
                (        <boolean_relation>
                |        'NOT' <boolean_factor>
                |        '(' <condition> ')'
                |        <function_expression>
                )
<boolean_relation > :=
                (        <expression > { ( '==' |  '<=' | '<' ) <expression> }
                |        <expression > { ( '==' |  '>=' | '>' ) <expression> }
                |        <expression>  '!= ' <expression>
                )
```

2.5 Link declarations

```
<link_declarations> :=          { <link_declaration> $ ';' } [';']
```

```
<link_declaration> :=
        (       'FOR' <link_count_spec> <indexed_link_declarations>
        'END_FOR'
        |       <simple_link_declaration>
        )
<link_count_spec> :=
        <counter_name> '=' '1' ',' <model_par_name>
<indexed _link_declaration> :=
                ( <link_type_name> | 'GENERIC' )
                <link_name> [ '[' <counter_name> ']' ]
                { <link_var_spec> $ ',' }
<simple_link_declaration> :=
                ( <link_type_name> | 'GENERIC' )
                <link_name> { <link_var_spec> $ ',' }
<link_var_spec> :=
                [ <pos_direction> ] <var_name> [ '[' <counter_name> ']' ]
<pos_direction> :=      ( 'POS_IN' | 'POS_OUT' )
```

2.6 Variable declarations

```
<var_decl> :=
                <quantity_type_spec> <var_name_spec> <var_role>
                [ <default_val> [ <min_val> <max_val> ] ] <description>
<var_name_spec> :=      <var_name> [ <field_decl> ]
<field_decl> :=         '[' { <field_size> $ ',' } ']'
<field_size>:=          <model_par_name>
<var_role> :=           ( 'IN' | 'OUT' | 'A_S' | 'LOC' )
<default_val> :=        <value>
<min_val> :=            ( <value> | <machine_limit> )
<max_val> :=            ( <value> | <machine_limit> )
<description> :=        <string>
```

2.7 Model parameter declarations

```
<model_par_decl> :=
                'INT' <model_par_name> <m_par_role>
                [ <model_def_val> [ <model_min_val> <model_max_val> ] ]
                <description>
<model_def_val> :=      <unsigned_integer>
<m_par_role> :=         ( 'SMP' | 'CMP' )
<model_min_val> :=      <unsigned_integer>
<model_max_val> :=      ( 'BIGINT' | <unsigned_integer> )
```

2.8 Parameter declarations

```
<par_decl> :=
                <quantitiy_type_spec> <par_name_spec> <par_role>
                [ <default_val> [ <min_val> <max_val> ] ] <description>
<par_name_spec> :=      <par_name> [ <field_decl> ]
<par_role> :=           ( 'S_P' | 'C_P' )
```

2.9 Function declarations

```
<function_decl> :=
                'FUNCTION' <type_spec> <function_name> '(' <formal_list> ')'
                'LANGUAGE' <language>
                ['INPUT'        { <formal_decl> } ]
                ['OUTPUT'       { <formal_decl> } ]
                <body>
                'END_FUNCTION'
<type_spec> :=                  ( 'INT' | 'FLOAT' | 'BOOLEAN' | 'VOID' )
<formal_list> :=                [ { <formal_name> $ ',' } ]
<language> :=                   ( 'NMF' | 'F77' | 'C')
<formal_decl> :=
                (       'INT' <formal_var_list> ';'
                |       'FLOAT' <formal_var_list> ';'
                |       'STRING' <formal_var_list> ';'
                )
<formal_var_list> :=            { <formal_var_spec> $ ', ' }
<formal_var_spec> :=            <formal_name> [ <formal_field_decl> ]
<formal_field_decl> :=          '[' { <formal_field_size> $ ',' } ']'
<formal_field_size> :=          <formal_name>
<body> :=       (       'EXTERNAL'  <external_name> '(' <formal_list> ')'
                |       'CODE'          <code>          'END_CODE'
                )
<code> :=       (       { <body_statement> $ ';' } [';']
                |       Code according to the specified non-NMF langauge
                )
<body_statement> :=  ( <statement> | 'RETURN' <expression> )
```

2.10 Low level constructs

2.10.1 Numbers

```
<value> :=                      ( <integer> | <real> | [ <sign> ] <const_name> )
<machine_limit> :=              [ <sign> ] ('BIG' | 'SMALL' | 'BIGINT' )
<unsigned_constant> :=
                ( <unsigned_integer> | <unsigned_real> | <const_name> )
<integer> :=                    [ <sign> ] <unsigned_integer>
<unsigned_integer> :=           { <digit> }
<real> :=                       [ <sign> ] <unsigned_real>
<unsigned_real> :=
                (       <fraction> [ <exponent> ]
                |       <unsigned_integer> <exponent>
                )
<fraction> :=
                (       <unsigned_integer> '.' [ <unsigned_integer> ]
                |       '.' <unsigned_integer>
                )
<exponent> :=                   'E' <integer>
<sign> :=                       ( '+' | '-' )
```

2.10.2 Names

```
<quantity_type_name> :=         <ident>
<link_type_name> :=             <ident>
```

```
<const_name> :=          <ident>
<counter_name> :=        <ident>
<link_name> :=           <ident>
<var_name> :=            <ident>
<model_name> :=          <ident>
<model_par_name> :       <ident>
<par_name> :=            <ident>
<function_name> :=       <ident>
<formal_name> :=         <ident >
<external_name> :=       <ident>
<ident> :=               <letter> [ { <id_char> } ]
```

2.10.3 Text strings

```
<string> :=              '"' [ { <char> } ] '"'
```

2.10.4 Characters

```
<id_char> :=             ( <letter> | <digit> | '_' | '$' )
<letter> :=
            (   'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J'
            |   'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T'
            |   'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' |
            |   'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j'
            |   'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't'
            |   'u' | 'v' | 'w' | 'x' | 'y' | 'z'
            )
<digit> :=
            ( '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0' )
<char> :=                ( <letter> | <digit> | <special_char> )
<special_char> :=
            (  '!'  |  '"'  |  '#'  |  '$'  |  '%'  |  '&'
            |  '('  |  ')'  |  '*'  |  '+'  |  ','  |  '-'  |  '.'  |  '/'
            |  ':'  |  ';'  |  '<'  |  '='  |  '>'  |  '?'
            |  '@'  |  '['  |  '\'  |  ']'  |  '^'  |  '_'
            |  '`'  |  '{'  |  '|'  |  '}'  |  '~'  |  ' '
            | <apostrophe> | <newline> | <tab>
            )
```

3 Semantic notes

The formal syntax above is permissive; it does not describe in detail the intended use of the NMF. It is reasonable to add some semantic rules to clarify the intentions.

Some of the rules given below could have been formulated in syntax, but the result would have been circumstantial and less legible.

1 All names used to identify quantities such as variables, parameters, etc., must be declared globally or locally.

2 Global names must be unique across all global declarations; i.e. a quantity type can not have the same name as a link type.

3 Names declared locally within a model must be unique across all local declarations, e.g. a link can not have the same name as a variable or parameter. Locally declared names and counter names may coincide with global ones; the local declarations have precedence. This is so, in order to make possible a normal extension of the global declarations.

4 Assignment of variables may only occur in the EQUATIONS section; only LOC and A_S variables may be assigned. Assignment of parameters and model parameters may only occur in the PARAMETER_PROCESSING section.

5 Any variable, parameter, or model parameter may only be assigned once. LOC variables and parameters must be assigned before they are referenced. A_S variables must be referenced before they are assigned.

6 The EQUATIONS section may contain references to variables, parameters, and model parameters. The PARAMETER_PROCESSING section may contain references to parameters and model parameters. Assignment of model parameters may only depend on other model parameters.

7 The number of OUT variables must agree with the number of equations.

8 All IN variables must appear in link declarations.

9 No LOC or A_S variables may occur in links.

10 All THRU variables but no CROSS variables should have direction specified by POS_IN or POS_OUT. GENERIC variables are always CROSS.

11 The PARAMETER_PROCESSING section cannot contain equations. Statements in the EQUATIONS section may consist of equations and assignments, but equations may not appear enclosed by IF - END_IF.

12 Counters are not declared separately. The same counter name may be used repeatedly.

13 Functions called in procedure calls must have type 'VOID'. Function expressions used in arithmetic expressions (section 2.4) must have type 'INT' or 'FLOAT', those used in Boolean factors (section 2.4.2) must have type 'BOOLEAN'.

14 Indexed variables are allowed in links, only when variable and link are vectors and indices are synchronized.

15 In function declarations, <external_name> must be the name of a procedure or function defined in the language specified.

16 A function body cannot contain equations.

17 Only IN and OUT variables may be used with derivatives.

18 All formal parameters must be declared as INPUT or OUTPUT or both.

19 Variables and parameters, which are declared as arrays (vectors or matrices), should be used with (the right number of) indices, except when they are used as arguments in function calls and the corresponding parameter is an array name. In this case, they should appear without indices and the corresponding parameter should be declared with the same size.

20 Formal parameters used as field sizes for formal arrays must be of type 'INT'.

21 A THRU variable in a model can appear in at most two links, one with POS_IN direction and one with POS_OUT direction.

22 The string of a description may not exceed 80 characters, each whitespace is counted as one character.

23 When an NMF function body is represented by an EXTERNAL call, the following applies:

- The name of the external routine may coincide with the name of any NMF-defined quantity. It may e.g. agree with the name of the calling NMF function.
- The type of the external routine is assumed to agree with the type of the NMF function. The types of parameters in the external call are assumed to agree with the corresponding (formal) parameters of the external routine. (No checking of these agreements will normally be possible when the NMF function is translated.)
- The parameters used in the external call must all appear among the formal parameters of the NMF function, but may appear in a different order. Each formal parameter may occur an arbitrary number of times, including zero, among the parameters of the external call. (As a corollary, we note that if constants or expressions are desired as parameters in the external call, the body must be reformulated as an NMF statement, calling a second NMF function.)

24 A non-integer value is rounded when it is assigned to a model parameter, or used as an index or an integer input argument of a function.

8. APPENDIX 2 EXAMPLES OF MODEL DEFINITIONS

The model examples shown in these Appendices have recently been translated via the ASHRAE translator and used in test runs with the IDA environment. Some caveats are in order, though:

- The validity ranges included in the NMF descriptions have not been tested properly.

- In some of the control components, variables have "natural" limits like the interval [0,1], etc. The event handling however, at least as it is implemented in IDA Solver, must be able to pass the event limit in order to detect it. It is still an open question how this conflict should be resolved.

CONTENTS

Global NMF Types		40
Example 1	Thermal Mass	42
Example 2	Homogeneous Wall	43
Example 3	Relay (Thermostat)	45
Example 4	Backlash (Actuator w hysteresis)	46
Example 5	Mean radiant zone	49
Example 6	Function U_film	51
Example 7	Zone with Bidirectional Air Paths	53
Example 8	Bidirectional Leak	54

Global NMF Types

```
QUANTITY_TYPES

/* Quantity types are used for both variables and parameters. For
   parameters the kind is irrelevant. */

/* CROSS = Potential, non-directional */
/* THRU = Flow, directional */

/* GENERAL and THERMODYNAMICS */
/* type name      unit                kind */
   Angle          "Deg"               CROSS
   Area           "m2"                CROSS
   Control        "dimless"           CROSS
   Density        "kg/m3"             CROSS
   Enthalpy       "J/kg"              CROSS
   Factor         "dimless"           CROSS
   FractFlow_h    "mg/h"              THRU
   Fraction       "kg/kg"             CROSS  /* pollutant/total mass */
   Fraction_y     "mg/kg"             CROSS
   HeatCap        "J/K"               CROSS
   HeatCapA       "J/(K m2)"          CROSS
   HeatCapM       "J/(kg K)"          CROSS
   HeatCond       "W/K"               THRU
   HeatCondL      "W/(m K)"           THRU
   HeatCondA      "W/(m2 K)"          THRU
   HeatFlux       "W"                 THRU
   HeatFlux_k     "kW"                THRU
   HeatFlux_M     "MW"                THRU
   HeatRes        "K/W"               CROSS
   HeatResA       "(m2 K)/W"          CROSS
   HumRatio       "kg/kg"             CROSS  /* water/dry air */
   HumRatio_m     "g/kg"              CROSS
   Length         "m"                 CROSS
   Mass           "kg"                CROSS
   MassFlow       "kg/s"              THRU
   MassFlow_h     "kg/h"              THRU
   MassFlow_y     "mg/s"              THRU
   OnOff          "dimless"           THRU
   Pressure       "Pa"                CROSS
   Pressure_k     "kPa"               CROSS
   Radiation      "W"                 THRU
   RadiationA     "W/m2"              THRU
   Temp           "Deg-C"             CROSS
   Temp_F         "Deg-F"             CROSS
   Temp_K         "K"                 CROSS
   Time           "s"                 CROSS
   Velocity       "m/s"               THRU
   ViscDyn        "kg/(m s)"          THRU
   ViscKin        "m2/s"              THRU
   Volume         "m3"                CROSS
   VolFlow        "m3/s"              THRU
   VolFlow_h      "m3/h"              THRU

/* OTHER APPLICATION FIELDS */
/* type name      unit                kind */
   Current        "A"                 THRU
   ElCap          "F"                 CROSS
   ElInduct       "H"                 CROSS
   ElRes          "Ohm"               CROSS
   Power          "W"                 THRU
   Voltage        "V"                 CROSS

LINK_TYPES
```

```
/* type name         variable types... */

   Q                  (HeatFlux)
   T                  (Temp)
   Z                  (Factor)
   X_y                (Fraction_y)
   X_h                (FractFlow_h)
   HQ                 (Enthalpy, HeatFlux)
   TQ                 (Temp, HeatFlux)
   TQRR               (Temp, HeatFlux, Radiation, Radiation)
   TR                 (Temp, RadiationA)
   RRRR               (RadiationA, RadiationA, RadiationA, RadiationA)

   M                  (MassFlow)
   MT                 (MassFlow, Temp)

   PM                 (Pressure, MassFlow)
   PMT                (Pressure, MassFlow, Temp)
   PMTQ               (Pressure, MassFlow, Temp, HeatFlux)

   MoistAir           (Pressure, MassFlow, Temp, HumRatio)
   VentX              (Pressure, MassFlow_h, Temp, Fraction_y)
   BidirFlow          (Pressure, MassFlow, Enthalpy, HeatFlux)
   BidirX             (Pressure, MassFlow_h, Temp, HeatFlux,
                       Fraction_y, FractFlow_h)
   HeatSun            (Temp, HeatFlux, RadiationA, RadiationA)

   ControlLink        (Control)
   ControlLimit       (Control, Control)

   UI                 (Voltage, Current)
```

```
CONSTANTS

/* name          value       unit           comment */

   ABS_ZERO      -273.16     "Deg-C"        /* absolute zero temp */
   BOLTZ         5.67E-8     "W/(m2 K4)"    /* Stefan Boltzmann */
   CP_AIR        1006.       "J/(kg K)"     /* air specific heat */
   CP_VAP        1805.       "J/(kg K)"     /* watervapor specific heat */
   CP_WAT        4187.       "J/(kg K)"     /* water specific heat */
   CV_AIR        720.        "J/(kg K)"     /* air specific heat */
   G             9.81        "m/s2"         /* gravity acceleration */
   GASCON        287.        ""             /* general gas constant */
   HF_VAP        2.501E6     "J/kg"         /* water vaporization heat */
   LAMBDA_AIR    0.0243      "W/(m K)"      /* air thermal conductivity */
   LAMBDA_WAT    0.554       "W/(m K)"      /* water thermal
                                               conductivity */
   P_ATM_0       1.013E5     "Pa"           /* standard air pressure */
   PI            3.1415927   "dimless"      /* the pi number */
   PRANDTL_AIR   0.71        "dimless"      /* air Prandtl number */
   RHO_AIR       1.2         "kg/m3"        /* air density */
   RHO_WAT       1000.       "kg/m3"        /* water density */
   VISC_WAT      1.E-3       "kg/(m s)"     /* water dynamic viscosity */
```

Example 1

```
CONTINUOUS_MODEL Tq_thermal_mass

ABSTRACT "Thermal mass with variable number of conduction links
          and variable number of heat sources controlled from outside"

EQUATIONS

/* heat balance */
   v_rho_cp * T' = SUM j = 1,n_cond Q[j] END_SUM
                 + SUM i = 1, n_src Gamma[i] * q_src[i] END_SUM
   GOOD_INVERSES (T) ;

LINKS

/* type                  name          variables... */

   FOR j = 1,n_cond
     TQ                  Cond[j]       T, POS_IN Q[j]
   END_FOR ;

   FOR i = 1,n_src
     ControlLink         Src_ctrl[i]   Gamma[i]
   END_FOR ;

VARIABLES

/* type      name           role def min      max  description */
   temp      T              OUT  0.  abs_zero BIG  "mass temperature"
   HeatFlux  Q[n_cond]      IN   0.  -BIG     BIG  "conducted heat"
   Control   Gamma[n_src]   IN   0   0        1    "source control (0,1)"


MODEL_PARAMETERS

/* type    name      role  def  min  max     description */
   INT     n_cond    SMP   1    0    BIGINT  "no of conduction links"
   INT     n_src     SMP   1    0    BIGINT  "no of controllable
                                              sources"

PARAMETERS

/* type      name         role def min   max  description */

/* easy access parameters */
   Volume    v            S_P  1.  SMALL BIG  "Volume"
   Density   rho          S_P  1.  SMALL BIG  "Density"
   HeatCapM  cp           S_P  1.  SMALL BIG  "Heat capacitivity (J/kg K)"
   HeatFlux  q_src[n_src] S_P  1.  SMALL BIG  "Heat source intensity"

/* derived parameters */
   HeatCap   v_rho_cp     C_P  1.  SMALL BIG  "Thermal mass"

PARAMETER_PROCESSING

   v_rho_cp := v * rho * cp ;


END_MODEL
```

Example 2

```
CONTINUOUS_MODEL tq_hom_wall

ABSTRACT
   "A 1D finite difference wall model.
    One homogeneous layer.
    TQ interfaces on both sides."

EQUATIONS

/* space discretized heat equation */

   c_coeff * T'[1] = Taa - 2.*T[1] + T[2] ;
   c_coeff * T'[n] = T[n - 1] - 2. * T[n] + Tbb ;

   FOR i = 2, (n-1)
     c_coeff * T'[i] = T[i - 1] - 2. * T[i] + T[i + 1];
   END_FOR ;

/* boundary equations */

   0 = -Ta + .5 * (Taa + T[1]) ;
   0 = -Tb + .5 * (T[n] + Tbb) ;
   0 = -Qa + d_coeff * (Taa - T[1]) ;
   0 = -Qb + d_coeff * (Tbb - T[n]) ;

LINKS

/*  type           name      variables ....   */

    TQ             a_side    Ta, POS_IN Qa ;
    TQ             b_side    Tb, POS_IN Qb ;

VARIABLES

/* type      name    role  def  min       max    description*/

   Temp      T[n]    OUT   0.   abs_zero  BIG    "temperature profile"
   Temp      Ta      OUT   0.   abs_zero  BIG    "a-side surface temp"
   Temp      Tb      OUT   0.   abs_zero  BIG    "b-side surface temp"
   Temp      Taa     OUT   0.   abs_zero  BIG    "a-side virtual temp"
   Temp      Tbb     OUT   0.   abs_zero  BIG    "b-side virtual temp"
   HeatFlux  Qa      IN    0.   -BIG      BIG    "a-side entering heat"
   HeatFlux  Qb      IN    0.   -BIG      BIG    "b-side entering heat"

MODEL_PARAMETERS

/* type      name    role  def  min       max    description   */
   INT       n       SMP   3    3         BIGINT "no of temp layers"
```

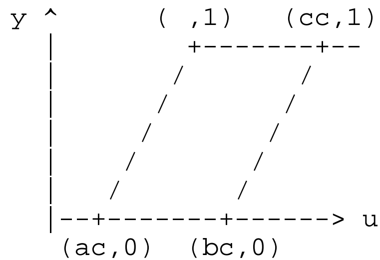
```
  PARAMETERS

/* type      name     role def   min   max  description  */

/* easy access parameters */
   Area      a        S_P  10.   SMALL BIG  "wall area"
   Length    thick    S_P  .25   SMALL BIG  "wall total thickness"
   HeatCondL lambda   S_P  0.5   SMALL BIG  "heat transfer coeff"
   Density   rho      S_P  2000. SMALL BIG  "wall density"
   HeatCapM  cp       S_P  900.  SMALL BIG  "wall heat capacity"
/* derived parameters */
   generic   d_coeff  C_P                   "lambda*a/dx"
   Length    dx       C_P                   "layer thickness"
   generic   c_coeff  C_P                   "rho*cp*dx*dx/lambda"

PARAMETER_PROCESSING

    dx := thick / n ;
    c_coeff := rho * cp * dx * dx / lambda;
    d_coeff := lambda * a / dx ;


END_MODEL
```

Example 3

```
CONTINUOUS_MODEL Thermostat

ABSTRACT "Thermostat w dead band tmin-tmax,
          OFF=0, ON=1 at low temperature"


EQUATIONS

    Out_signal = Old_signal
    BAD_INVERSES () ;

    Top := eventp(Hi, T - tmax) ;
    Bottom := eventn(Lo, T - tmin) ;
    Old_signal := IF Top > 0 THEN
                     0
                  ELSE_IF Bottom < 0 THEN
                     1
                  ELSE
                     Old_signal
                  END_IF ;


LINKS

/* type         name         variables       */

   ControlLink  Out_sign     Out_signal ;
   ControlLink  In_sign      T ;

VARIABLES

/* type  name          role def  min  max  description    */

   generic  Hi         A_S                 "Event var up"
   generic  Lo         A_S                 "Event var down"
   generic  Old_signal A_S                 "State On or Off"
   generic  T          IN                  "In-signal"
   generic  Out_signal OUT  0    0    1    "Out-signal On=1 or Off=0"
   generic  Top        LOC                 "Over the top"
   generic  Bottom     LOC                 "Down under"

PARAMETERS

/* type    name        role def  min  max  description    */

   temp    tmin        S_P                 "lower limit of dead band "
   temp    tmax        S_P                 "higher limit of dead band"


END_MODEL
```

Example 4

```
CONTINUOUS_MODEL  Hysteresis

ABSTRACT "Hysteresis, output varying continuously between 0 and 1,
          sloping sides; limiting romboid:
          upper left (1-a-b,1)  upper right (1-a,1)
          lower left (a,0)      lower right (a+b,0)."

/*
```




```
         ac = a, bc = a+b, cc = 1-a
*/

EQUATIONS

   Y = IF  U > U_old  THEN
         IF S_right < 0 THEN
           Y_old
         ELSE_IF S_top < 0 THEN
           (U-bc) * slope
         ELSE
           1
         END_IF
       ELSE
         IF S_left > 0 THEN
           Y_old
         ELSE_IF S_bottom > 0 THEN
           (U - ac) * slope
         ELSE
           0
         END_IF
       END_IF
   BAD_INVERSES (U) ;

   U_set = IF U > cc THEN
             cc
           ELSE_IF U < ac THEN
             ac
           ELSE
             U
           END_IF
   BAD_INVERSES (U) ;

/* update assigned states */

   IF U > U_old THEN

     IF event (S_right, U - bc - Y_old/slope) < 0  THEN

/*   prepare for next regime */
       S_top := U - cc
     ELSE
       Loc := event (S_top, U - cc)
     END_IF ;
```

```
/*   prepare for change of direction */
     S_left := U - ac - Y_old/slope ;
     S_bottom := U - ac ;

   ELSE
     IF event (S_left, U - ac - Y_old/slope) > 0  THEN

/*   prepare for next regime */
       S_bottom := U - ac
     ELSE
       Loc := event (S_bottom, U - ac)
     END_IF ;

/*   prepare for change of direction */
     S_right := U - bc - Y_old/slope ;
     S_top := U - cc ;
   END_IF ;

   IF  U > bc + Y_old/slope
   OR  U < ac + Y_old/slope  THEN
     Y_old := Y
   END_IF ;

   U_old := U ;

LINKS

/* type              name         variables     */

   ControlLimit      In_signal    U, U_set ;
   ControlLink       Out_signal   Y ;

VARIABLES

/* type   name      role  def  min  max  description    */

 generic  S_bottom  A_S                  "Event var left side to bottom"
 generic  S_left    A_S                  "Event var reaching left side"
 generic  S_right   A_S                  "Event var reaching right side"
 generic  S_top     A_S                  "Event var right side to top"
 generic  U         IN                   "In-signal"
 generic  U_old     A_S                  "Old in-signal saved"
 control  U_set     OUT                  "Limited in-signal, fed back"
 control  Y         OUT                  "Out-signal"
 generic  Y_old     A_S                  "Old out-signal saved"
 generic  Loc       LOC                  "Dummy"
```

```
PARAMETERS

/* type    name    role def  min  max    description    */

/* easy access parameters   */
 control   a       S_P  0    0    0.5    "lower left corner = (a,0)"
 control   b       S_P  0.5  0    1      "dead band = width of romboid"

/* derived parameters   */
 control   ac      C_P  0    0    1      "lower left corner  = (ac,0)"
 control   bc      C_P  0.5  0    1      "lower right corner = (bc,0)"
 control   cc      C_P  0.5  0    1      "upper right corner = (cc,1)"
 control   ramp    C_P  0.5  0    1      "width of sides"
 control   slope   C_P  2    1    BIG    "slope of sides"

PARAMETER_PROCESSING

   ac := a ;
   cc := 1. - a ;
   ramp := 1. - 2*a - b ;

   IF ramp > 1E-4 THEN
      bc := a + b ;
   ELSE
      ramp := 1E-4 ;
      bc := 1 - a - ramp ;
      IF bc < ac THEN
         CALL nmf_error ("Wrong parameters in hysteresis (1-2*a-b<0)")
      END_IF ;
   END_IF ;

   slope := 1. / ramp ;


END_MODEL
```

Example 5

```
CONTINUOUS_MODEL M_rad_zon

ABSTRACT
  " A zone model with a variable number of surfaces.

    Longwave radiation is treated with Stefan-Boltzmann
    law between surface and mean radiant temperature.

    Nonlinear convective heat transfer between surface and air.

    Only one source of shortwave radiation (window).

    Ventilation and infiltration are supposed to be modelled
    in a separate component, connected by a TQ link."

EQUATIONS

/* calc film coefficients */
   FOR i = 1, n_surf
     U_f[i] :=
       IF LINEARIZE(1) THEN
          1.
       ELSE
          U_film (T_zone, T_surf[i], surf_slope[i], T_dif[i])
       END_IF
   END_FOR ;

/* heat from people */
   Q_person := IF T_zone < 20. THEN q_activity
               ELSE_IF T_zone > 40. THEN 0
               ELSE q_activity * (40. - T_zone) / 20.
               END_IF ;
   Q_active := N_people * Q_person ;

/* air heat balance */
   0 = Q_vent + Q_active + Q_zone +
         SUM j = 1, n_surf
           surf_area[j] * U_f[j] * (T_surf[j] - T_zone)
         END_SUM
   BAD_INVERSES () ;

/* long wave radiation balance */
   0 = SUM i = 1, n_surf
         surf_area[i] * ((T_surf[i]-ABS_ZERO)**4
                         -(M_rad_t-ABS_ZERO)**4)
       END_SUM
   BAD_INVERSES () ;

/* surface heat balances */
   FOR i = 1, n_surf
     0 = Q_surf[i]
       + surf_area[i] * U_f[i] * (T_zone - T_surf[i])
       + q_frac[i] * Q_source
       + 0.93 * BOLTZ * surf_area[i]
       * ((M_rad_t-ABS_ZERO)**4 - (T_surf[i]-ABS_ZERO)**4)
     BAD_INVERSES (Q_source) ;
   END_FOR ;
```

```
LINKS

/* type           name             variables ....  */

   TQ             zone_air         T_zone, POS_IN Q_vent ;
   Q              sw_source        POS_IN Q_source ;
   Q              direct           POS_IN Q_zone ;
   Z              occupancy        N_people ;

   FOR i = 1, n_surf
     TQ           surface[i]       T_surf[i], POS_IN Q_surf[i]
   END_FOR ;

VARIABLES

/* type     name             role def min      max  description*/

Temp        T_zone           OUT  0.  abs_zero BIG  "zone temperature"
HeatFlux    Q_vent           IN   0.  -BIG     BIG  "net heat flow to air
                                                     (from separate model)"
Factor      N_people         IN   0.  0.       BIG  "number of people"
HeatFlux    Q_zone           IN   0.  -BIG     BIG  "heat direct to air"
Temp        M_rad_t          OUT  0.  abs_zero BIG  "mean radiant temp"
Temp        T_surf[n_surf]   IN   0.  abs_zero BIG  "surface temperature"
HeatFlux    Q_surf[n_surf]   OUT  0.  -BIG     BIG  "surface heat flow"
HeatFlux    Q_source         IN   0.  -BIG     BIG  "short wave
                                                     radiation"
HeatFlux    Q_active         LOC                    "heat flow fr people"
HeatFlux    Q_person         LOC                    "heat flow / person"
Temp        T_dif[n_surf]    LOC                    "saved temp diff"
HeatCondA   U_f[n_surf]      LOC                    "film coeff"

MODEL_PARAMETERS

/* type        name           role   def min max     description*/

   INT         n_surf         SMP    6   1   BIGINT  "number of surfaces"

PARAMETERS

/* type   name                 role  def min max   description*/

Heatflux  q_activity           S_P   0.  0.  BIG   "activity / person"

/* easy access parameters for rectangular surfaces */
Length    surf_height[n_surf]  S_P   2.4 0.  BIG   "surface height"
Length    surf_width[n_surf]   S_P   3.  0.  BIG   "surface width"
Angle     surf_slope[n_surf]   S_P   0.  0.  180.  "surface slope,
                                                    0=floor,
                                                    180=ceiling"
Factor    q_frac[n_surf]       S_P   .1  0.  BIG   "surface shortw share"

/* derived parameters */
Area      surf_area[n_surf]    C_P   1.  0.  BIG   "surface area"

PARAMETER_PROCESSING

   FOR i = 1, n_surf
     surf_area[i] := surf_height[i] * surf_width[i]
   END_FOR ;

END_MODEL
```

Example 6

```
FUNCTION FLOAT U_film (TAir, TSurf, SurfSlope)

/* Calculate film coefficient for room surfaces */

LANGUAGE F77

INPUT
      FLOAT TAir, TSurf, SurfSlope;

CODE
      REAL FUNCTION U_FILM (TAir, TSurf, SurfSlope)

      REAL TAir, TSurf, SurfSlope

* Calculate surface film coefficient U_FILM as a function
* of temperature difference (TAir - TSurf) and surface slope
* given as SurfSlope=0 for floor and SurfSlope=180 for ceiling.

      REAL ALFA_HOR(2, 8), ALFA_VERT(2, 8)
      COMMON /ALFA_TAB/ ALFA_HOR, ALFA_VERT

      INTEGER I2
      REAL ABS_DIF, ALFA, DIF, GRAD
      EXTERNAL TABLE


      I2 = 2
      DIF = TAir - TSurf
      IF (SurfSlope .LT. 89.) THEN
*   Floor
         CALL TABLE (8, ALFA_HOR, I2, -DIF, ALFA, GRAD)

      ELSE IF (SurfSlope .GT. 91.) THEN
*   Ceiling
         CALL TABLE (8, ALFA_HOR, I2, DIF, ALFA, GRAD)

      ELSE
*   Wall
         ABS_DIF = ABS (DIF)
         CALL TABLE (8, ALFA_VERT, I2, ABS_DIF, ALFA, GRAD)

      ENDIF

      U_FILM = ALFA
      RETURN
      END

* For completeness the subroutine TABLE follows even though it may
* well be stored somewhere else, e.g. in a fortran library etc.

   SUBROUTINE TABLE (NCOL, TableVal, TABLREF, ARG, VAL, SLOPE)

      INTEGER NCOL, TABLREF
      REAL TableVal(2, NCOL), ARG, VAL, SLOPE

* INTERPOLATE LINEARLY IN TABLE 'TableVal',
* FIND VALUES 'VAL' & 'SLOPE' CORRESPONDING TO ARGUMENT 'ARG'.
* 'TABLREF' POINTS INTO TABLE AND IS ADJUSTED,
* UP OR DOWN, UNTIL A SUITABLE INTERVAL IS FOUND.
* IT IS ASSUMED THAT OUTMOST SEGMENTS STRETCH TO +- INFINITY.

      REAL T1,T2,V1,V2
```

```
      T1 = TableVal(1, TABLREF)

      IF (ARG .LT. T1) THEN
   20    CONTINUE
         IF (TABLREF .GT. 1) THEN
            TABLREF = TABLREF - 1
            T2 = TableVal(1, TABLREF)
            IF (ARG .LT. T2) THEN
               T1 = T2
               GOTO 20
            ENDIF
         ENDIF
         V1 = TableVal(2, TABLREF+1)
      ELSE
   40    IF (TABLREF .LT. NCOL) THEN
            TABLREF = TABLREF + 1
            T2 = TableVal(1, TABLREF)
            IF (ARG .GT. T2) THEN
               T1 = T2
               GOTO 40
            ENDIF
         ENDIF
         V1 = TableVal(2, TABLREF-1)
      ENDIF
      V2 = TableVal(2, TABLREF)
      SLOPE = (V2 - V1) / (T2 - T1)
      VAL = (ARG - T1) * SLOPE + V1

      RETURN
      END
END_CODE

END_FUNCTION
```

Example 7

```
CONTINUOUS_MODEL  Bdzone

ABSTRACT
  "A static zone model for air-exchange modelling. Bidirectional
   transports of energy plus a mass fraction are modelled."

EQUATIONS

/* mass conservation */
   0 = M_0 + SUM i=1, n M[i] END_SUM
   BAD_INVERSES () ;

/* energy conservation */
   0 = Q_zone + Q_0 + SUM i2=1, n Q[i2] END_SUM
   BAD_INVERSES () ;

/* fraction conservation */
   0 = xf_source + Xf_0 + SUM i3=1, n Xf[i3] END_SUM
   BAD_INVERSES () ;

LINKS

/* type   name         variables... */

BidirX    terminal_0   P, POS_IN M_0, T, POS_IN Q_0, X, POS_IN Xf_0;

FOR i = 1, n
  BidirX  terminal[i]  P, POS_IN M[i], T, POS_IN Q[i], X, POS_IN Xf[i]
END_FOR ;

TQ        air_temp     T, POS_IN Q_zone;

VARIABLES

/* type         name   role  def  min  max  description */
   MassFlow_h   M_0    OUT                  "terminal 0 massflow"
   MassFlow_h   M[n]   IN                   "terminal i massflow"
   Pressure     P      IN                   "zone floor level press"
   HeatFlux     Q_0    OUT                  "terminal 0 HeatFlux"
   HeatFlux     Q[n]   IN                   "terminal i HeatFlux"
   Temp         T      IN                   "zone temperature"
   FractFlow_h  Xf_0   OUT                  "terminal 0 transport"
   FractFlow_h  Xf[n]  IN                   "terminal i transport"
   Fraction_y   X      IN                   "zone fraction"
   HeatFlux     Q_zone IN                   "heat gain/loss in zone"

MODEL_PARAMETERS

/* type   name  role  def  min  max     description */
   INT    n     SMP   1    1    BIGINT  "Number of links minus one"

PARAMETERS

/* type        name      role  def min   max  description */
   Length      za        S_P                  "zone floor height
                                              from ground"
   Length      h         S_P   2.4 SMALL BIG  "zone height"
   Area        a         S_P   10  SMALL BIG  "zone floor area"
   FractFlow_h xf_source S_P   0   -BIG  BIG  "Mass fract source (or sink)"

END_MODEL
```

Example 8

```
CONTINUOUS_MODEL Bdleak

ABSTRACT "A powerlaw leak model w/ bidirectional transports of
          energy and a mass fraction."

EQUATIONS

/* driving pressure difference*/

   Rho1 := rho_20*(293/(T1 + 273)) ;
   Rho2 := rho_20*(293/(T2 + 273)) ;

   Dp := P1 - Rho1*G*zr1 - (P2 - Rho2*G*zr2) - Rho*G*dz ;

/* powerlaw massflow equation */

   M / 3600 = IF LINEARIZE (1) THEN c * Dp
              ELSE_IF abs (Dp) < dp0 THEN c0 * Dp
              ELSE_IF Dp > 0 THEN c * Dp**n
              ELSE -c * (-Dp)**n
              END_IF
   BAD_INVERSES () ;

/* convected heat through leak*/

   Q = IF LINEARIZE (1) THEN (T1 - T2) / 2
       ELSE_IF M > 0.0 THEN cp * T1 * M/3600
       ELSE cp * T2 * M/3600
       END_IF
   GOOD_INVERSES (Q) ;

/* fraction transported through leak*/

   Xf = IF LINEARIZE (1) THEN (X1 - X2) / 2
        ELSE_IF M > 0.0 THEN X1 * M/3600
        ELSE X2 * M/3600
        END_IF
   GOOD_INVERSES (Xf) ;

/* density of crack air is an assigned state (due to hysteresis) */

   Rho :=  IF LINEARIZE (1) THEN rho_20
           ELSE rho_20*(293/(IF EVENT(Mm, M) > 0 THEN T1
                             ELSE T2
                             END_IF + 273))
           END_IF ;

LINKS

/* type  name         variables... */

BidirX   terminal_1   P1, POS_IN  M, T1, POS_IN  Q, X1, POS_IN  Xf ;
BidirX   terminal_2   P2, POS_OUT M, T2, POS_OUT Q, X2, POS_OUT Xf ;

VARIABLES

/* type      name role def  min      max  description */

Density      Rho  A_S  1.2  .5       3    "density of leak air"
Density      Rho1 LOC  1.2  .5       3    "density of neighb.1 air"
Density      Rho2 LOC  1.2  .5       3    "density of neighb.2 air"
MassFlow_h   M    OUT  0    -BIG     BIG  "massflow through leak"
Pressure     P1   IN   1    -BIG     BIG  "terminal 1 pressure"
```

```
Pressure      P2   IN   2    -BIG     BIG  "terminal 2 pressure"
Temp          T1   IN   20   ABS_ZERO BIG  "Temperature of neighbor 1"
Temp          T2   IN   20   ABS_ZERO BIG  "Temperature of neighbor 2"
HeatFlux      Q    OUT  0    -BIG     BIG  "heat moved by massflow"
Fraction_y    X1   IN   .1   0        1    "fraction of neighbor 1"
Fraction_y    X2   IN   .1   0        1    "fraction of neighbor 2"
FractFlow_h   Xf   OUT  0    -BIG     BIG  "fraction moved by
                                            massflow"
MassFlow_h    Mm   A_S                     "massflow memory"
Pressure      Dp   LOC                     "effective pressure
                                            difference"


PARAMETERS

/* type   name     role def  min   max  description */

/* easy access parameters */
/* priority order: ela, c_t */

Generic   c_t      S_P  1    0     BIG  "powerlaw coeff [kg/(s Pa**n)]"
Generic   n        S_P  .5   .5    1.0  "powerlaw exponent [dimless]"
Area      ela      S_P  0    0     BIG  "equivalent leakage area
                                        at Dp=4 Pa (C_d = 1)"
length    dz       S_P  0               "rise fr terminal_1 to 2
                                        (may be < 0)"
length    zr1      S_P  0    0     BIG  "leak height from floor of
                                         neighb. 1"
length    za1      S_P                  "absolute floor level of
                                         neighb. 1"
length    za2      S_P                  "absolute floor level of
                                         neighb. 2"
HeatCapM  cp       S_P  1006 .5E3  3E3  "air cp"
Pressure  dp0      S_P  .1   SMALL BIG  "limit for linear flow"
Density   rho_20   S_P  1.2  SMALL 1.3  "density at ground press"

/* derived parameters, c is derived if ela > 0*/
Generic   c        C_P                  "powerlaw coeff [kg/(s Pa**n)]"
Generic   c0       C_P                  "linear coefficient"
length    zr2      C_P                  "leak height fr floor of
                                         neighb 2"

PARAMETER_PROCESSING

   zr2 := za1 + zr1 + dz - za2 ;

   c := IF ela > SMALL THEN
           ela * SQRT(4 * 2 * rho_20)/4**n
        ELSE
           c_t
        END_IF ;

   c0 := c * dp0**(n-1) ;


END_MODEL
```

9. APPENDIX 3 CHANGES IN VERSION 3

3.1 Syntax changes

1 Parameter declarations (section 2.8) have the same format as variable declarations (section 2.6). Parameters thus have a 'role', which can take values `S_P` or `C_P` (Supplied vs. Computed parameters). Normal input parameters are given the role `S_P`. Parameters that get their values defined by the parameter processing are given the role `C_P`. The role information makes it possible to translate the model without delving into functions called by the parameter processing.

2 Model parameters (section 2.7) also get the same basic format, but with roles `SMP` and `CMP` (Supplied vs. Computed Model Parameters). The same reasoning applies.

3 Variables, model parameters, and parameters all have the same syntax for default values and min/max values: [`def` [`min max`]]. This means that you can give, either none of the values, or just default, or all three values. It is recommended that default values, when that is meaningful, are specified in such a manner that they constitute a reasonable set of test data for the model. This should be taken to mean: with parameters and `IN` variables set to default, the model should deliver `OUT` variable values equal to their defaults.

4 Sections `VARIABLE_TYPES` and `PARAMETER_TYPES` in global declarations (section 2.1) have been combined into one section `QUANTITY_TYPES`. The original report required variable and parameter types to be different. This created a lot of impractical duplication and naming problems, since many physical quantities will often appear both as variables and parameters. The new types are used for both variables and parameters.

5 Concerns function declarations (section 2.9).

5.1 Lists of formal `INPUT` and `OUTPUT` declarations may be void (the keyword is omitted).

5.2 Function declarations are terminated by `END_FUNCTION`.

5.3 Function type '`BOOLEAN`' has been added.

5.4 Actual parameters can be of type `STRING`.

5.5 Function 'body' may be given in the form of NMF assignments, or as a call of an `EXTERNAL` routine, or (as before) as a complete routine in the specified language.

5.6 A `RETURN` statement can be used in NMF code in a function body. For non-VOID functions, the return statement defines the function value.

6 Some further semantic notes have been added (section 3, notes 13-24).

3.2 Revision of Global NMF Types

QUANTITY_TYPES

One of the syntax changes described in Appendix 3.1 was to replace the two sections `VARIABLE_TYPES` and `PARAMETER_TYPES` by one section `QUANTITY_TYPES`, valid for both variables and parameters.

In the previous version of the report, variable types and parameter types were required to have different names. Practical experience from modelling with NMF showed that this led to unnecessary duplication and naming problems. It is also a fact that a quantity may well be changed from parameter to variable status, or vice-versa, during the development of a model, without in essence changing quality. There were thus good reasons to amalgamate the two sections.

The type lists in the previous version were the result of uncoordinated contributions from several users and lacked consistency. Since the merging of the two lists required major changes anyhow, there was not much point in trying to keep any of the old typing. Instead, an effort was made to build the new list in a systematic way, and suggest some suitable mechanisms for later extensions.

Names are first chosen for types defined with basic SI units. To save space, inherent structure in these names is marked by use of case, rather than by inserted separating underscores. Examples:

```
Length       "m"
MassFlow     "kg/s"
HeatCond     "W/K"
```

New type names can be constructed by adding suffixes to those already defined. One important case is when types differ only by scaling (`kPa`, `MJ`, etc). Here, the underscores are used to increase legibility:

```
   Scale factors

_k   10E3       _m   10E-3
_M   10E6       _y   10E-6
_G   10E9       _n   10E-9
_T   10E12      _p   10E-12
```

Examples:

```
Enthalpy     "J/kg"            HeatFlux     "W"
Enthalpy_k   "kJ/kg"           HeatFlux_M   "MW"
```

Another case is when types can reasonably be derived from already defined types by making them specific, per unit length, per unit area, etc.

```
HeatCond     "W/K"             HeatCap    "J/K"
HeatCondL    "W/(K)"           HeatCapM   "J/(kg K)"
HeatCondA    "W/(m2 K)"
```

The current list of QUANTITY_TYPES is found in Appendix 2

LINK_TYPES

Link types can be given descriptive long names, e.g. MoistAir, BidirFlow, etc. However, since combinations of derived units tend to proliferate, the list of mnemonic names might soon become too long to be practical. To avoid this, systematic link names can be built from 'code' letters. The letters are chosen to indicate the constituent variables in proper order; the possible derived units can also be shown. For the code letters the following abbreviations are used:

```
C    Control          T    Temp_C
X    Fraction         V    VolFlow
G    Humidity         Z    Factor
H    Enthalpy
K    Temp K
M    MassFlow
P    Pressure         I    Current
R    Radiation        P    Power
Q    HeatFlux         U    Voltage
S    Entropy
```

Preferred order:
- Corresponding potential/flow adjacent in that order (PM, HQ, etc);
- Main flow first, then transported qualities.

Links containing variables in derived units use an extension of the basic letter combination to indicate scaling factors.

Examples:

```
TQ          (Temp, HeatFlux)
TQ$_k       (Temp, HeatFlux_k)
PMTX$k__y   (Pressure_k, MassFlow, Temp, Fraction_y)
```

Proliferation of type names should be avoided as much as possible. It is desirable that checking of link types be made at link level to retain the spirit of NMF, i.e. to achieve compatibility via strict typing. For the same reason, alternative link types, differing only in scaling (..$__), should not be introduced carelessly. Rather, for each new combination of variable types, required on an interface, a careful choice of the scaling factors should be made. Hopefully, this choice should be acceptable for other model developers as well.

The current list of QUANTITY_TYPES is found in Appendix 2

10. APPENDIX 4 A WORKED SYSTEM EXAMPLE

In the following appendix is shown a small system, built from components documented in this appendix. The system has been simulated with IDA, and some result plots are presented.

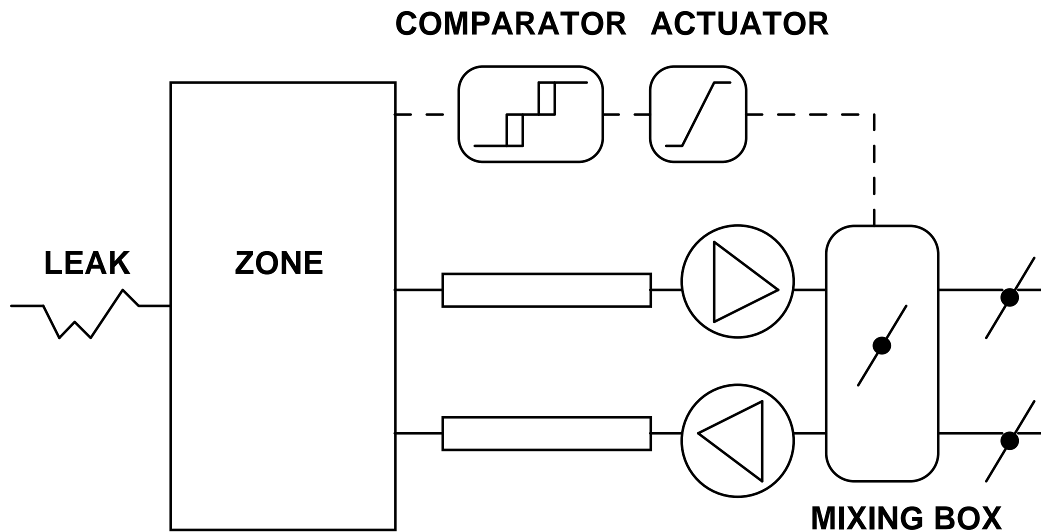
Contents

Demand Controlled Ventilation - System and Results		59
Example 1	Actuator Ramp with Discrete Input	62
Example 2	Comparator with Two Dead Bands	63
Example 3	Zone with Dynamic Concentration of Contaminant	65
Example 4	Leak between Zone and Ambient	67
Example 5	Fan	69
Example 6	Exhaust Terminal in Zone	73
Example 7	Supply Terminal in Zone	75
Example 8	Mixing Box	77

Demand Controlled Ventilation - System and Results

The system consists of a zone with supply and exhaust ventilation plus return air via a mixing box (figure 1).

Figure 1



The aim is to keep carbon dioxide concentration in the inhabited zone between 800 and 1000 ppm by controlling the amount of outside air through the mixing box. The amount of exhaust air is somewhat higher than the supply, the balance comes from infiltration. The CO_2 load from people follows an office type of pattern.

The supply and exhaust ducts have been reduced to one single component each, where the relevant pressure drops can be specified. The duct system, including fans and mixing box, etc, model one-directional flow, with links sufficient for that purpose. The leak paths require bidirectional flow modelling and use more elaborate links. The interface between these two different model families is supplied by zone terminals, with different plugs at either end.

The fan curves have been chosen quite steep in the pertinent area, so the net supply and exhaust flows are quite insensitive to the operation of the mixing box.

The control equipment consists of a comparator plus an actuator. The comparator checks for the two CO2 limits. As long as CO2 is between the limits the actuator is left at rest. When CO2 goes out of bounds, the actuator is instructed to ramp up or down at a constant pace. The ramping is terminated when CO2 again falls between limits, or alternatively, when the actuator reaches one of its limits.

The mixing box moves three dampers in parallel, in the return path and in the two paths leading outdoors. The signal from the actuator is interpreted as the desired fraction of outdoor air to put into the supply duct. The damper 'position' is assumed to translate to a linear variation in the flow characteristics of the dampers.

The models used are listed below, together with their parameter values, when these are different from the default values in the NMF models.

```
ZONE
TYPE BdZonT
  a       = 1300
  h       = 2.6
  xfrate  = 9.8

MIXING BOX
TYPE VxMix
  coutmin = .01
  coutmax = 2.
  cretmin = .01
  cretmax = 2.

EXHAUST
TYPE VxExhT
  c_t     = 7.

EXHAUST FAN
TYPE VxFan
  dp_nom[1] = 1000.  vf_nom[1] =     0
  dp_nom[2] =  500.  vf_nom[2] = 19000
  dp_nom[3] =    0.  vf_nom[3] = 21000

SUPPLY
TYPE VxSupT
  c_t     = 7.

SUPPLY FAN
TYPE VxFan
  dp_nom[1] = 1000.  vf_nom[1] =     0
  dp_nom[2] =  500.  vf_nom[2] = 18387
  dp_nom[3] =    0.  vf_nom[3] = 20322

LEAK
TYPE UtLeak
  c_t     = .071

COMPARATOR
TYPE CoCom2
  maxhi = 1000
  maxlo = 990
  minhi = 810
  minlo = 800

ACTUATOR
TYPE CoRamp
  slope = 4.
```

The simulation performed shows the first 24 hours after a start with outdoor air concentration in the zone (figures 2 and 3).

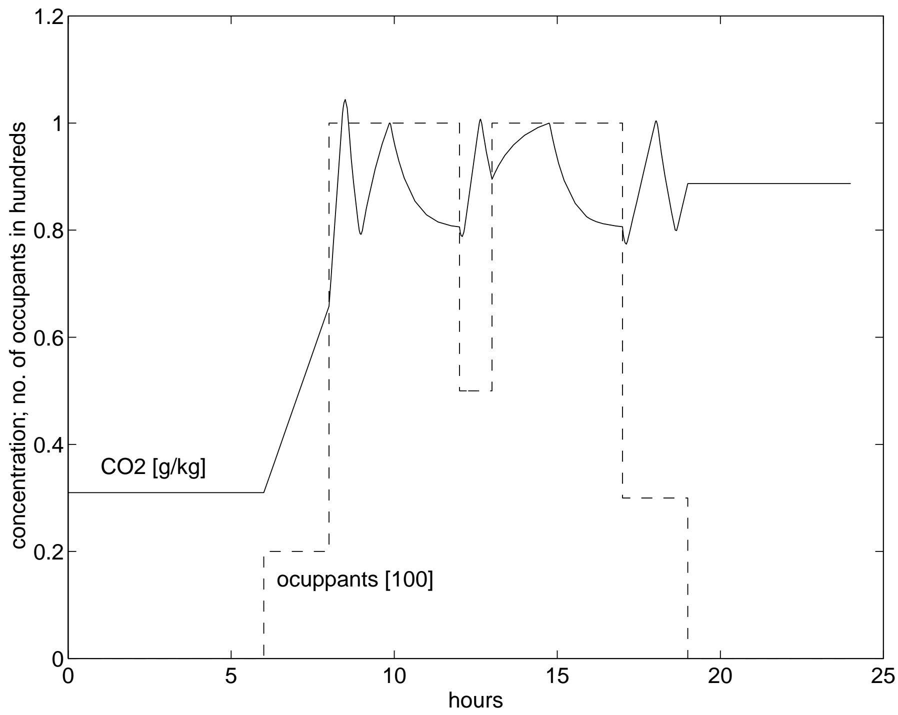


Figure 2

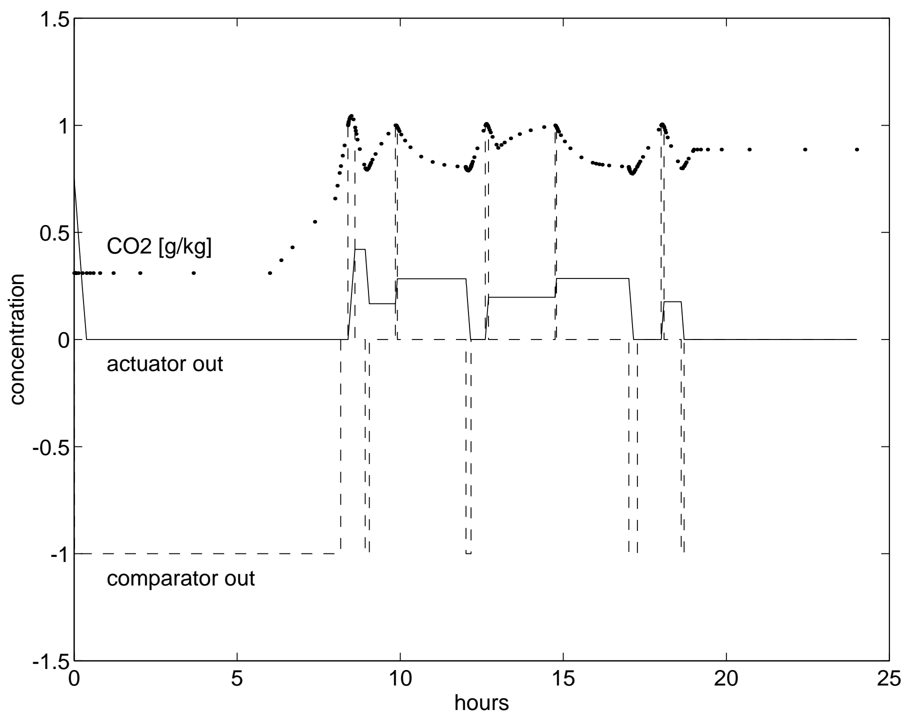


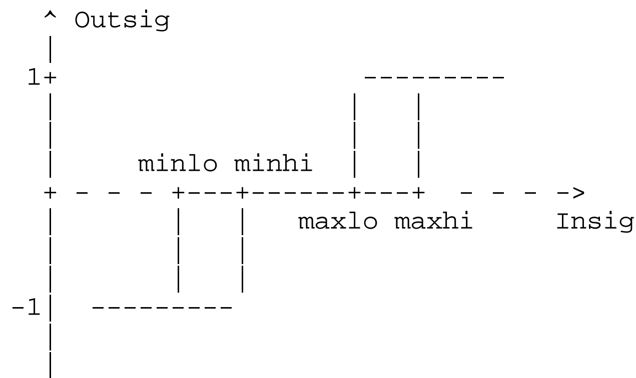
Figure 3

Example 1

```
CONTINUOUS_MODEL CoRamp

ABSTRACT "Actuator ramp with discrete input.
          While control input is 0, output remains the same.
          If control input is 1, output ramps up.
          If control input is -1, output ramps down.
          Output stays in the interval (lo,hi)."

EQUATIONS

/* RampOK allows movement when outsignal within limits;
   Insignal is one of -1, 0, 1 */
   Level' = slope*Insignal*RampOK
   GOOD_INVERSES (Level) ;

/* convert normalized output in (-1,1) to specific in (lo,hi) */
   OutLevel := lo * (1 - Level) / 2 + hi * (Level + 1) / 2 ;

/* Ensure that signal strictly in interval [-1,1] */
   OutSignal = IF OutLevel > hi THEN hi
               ELSE_IF OutLevel < lo THEN lo
               ELSE OutLevel
               END_IF
   GOOD_INVERSES (OutSignal) ;

   RampOK := IF event(Outside, Level * Insignal - 1) > 0 THEN
                0
             ELSE
                1
             END_IF;

LINKS
/* type          name         variables        */

   controllink   In_sig       Insignal ;
   controllink   Out_sig      Outsignal ;

VARIABLES
/* type   name      role def   min  max  description    */

 control  Outside   A_S  -0.5            ">0 if level outside (-1,1)"
 control  RampOK    A_S  0     0    1    "State =1 if ramping allowed,
                                                =0 otherwise"
 control  Insignal  IN   0     -1   1    "Input is -1, 0, or 1"
 control  Level     OUT  0.5   -1   1    "Normalized output in (-1,1)"
 control  OutLevel  LOC                  "Specific output in (lo,hi)"
 control  Outsignal OUT                  "Ditto strictly in (lo,hi)"

PARAMETERS
/* type   name      role def   min  max   description    */

 generic  slope     S_P  1.    -BIG BIG   "change of output level
                                             / time_unit"
 generic  lo        S_P  0.    -BIG BIG   "lower limit of output"
 generic  hi        S_P  1.    -BIG BIG   "upper limit of output"

END_MODEL
```

Example 2

```
CONTINUOUS_MODEL CoCom2

ABSTRACT "Comparator w 2 dead bands minlo-minhi maxlo-maxhi,
          outsig -1, 0, 1 below, between, above min & max"

/*
```




```
*/

EQUATIONS

   Outsig = Oldsig
   BAD_INVERSES () ;

   UpMax := eventp(HiMax, Insig - maxhi) ;
   DownMax := eventn(LoMax, Insig - maxlo) ;

   UpMin := eventp(HiMin, Insig - minhi) ;
   DownMin := eventn(LoMin, Insig - minlo) ;

   Oldsig := IF UpMax > 0 THEN
                1
             ELSE_IF DownMax < 0 AND UpMin > 0 THEN
                0
             ELSE_IF DownMin < 0 THEN
                -1
             ELSE
                Oldsig
             END_IF ;

LINKS

/* type          name          variables       */

   generic       In_sig        Insig ;
   ControlLink   Out_sig       Outsig ;
```

```
VARIABLES

/* type       name       role def  min  max    description    */

 generic     HiMax    A_S                     "Event var up"
 generic     HiMin    A_S                     "Event var down"
 generic     LoMax    A_S                     "Event var up"
 generic     LoMin    A_S                     "Event var down"
 generic     Oldsig   A_S   0    -1   1      "State -1, 0, 1"
 generic     Insig    IN                      "In-signal"
 generic     Outsig   OUT   0    -1   1      "Out-signal Lo=-1,
                                              Hi=1, Dead=0"
 generic     DownMax  LOC                     "Below max band"
 generic     DownMin  LOC                     "Below min band"
 generic     UpMax    LOC                     "Above max band"
 generic     UpMin    LOC                     "Above min band"

PARAMETERS

/* type    name      role  def  min  max  description    */

 generic   maxhi     S_P   2              "hi limit of max dead band"
 generic   maxlo     S_P   1.9            "lo limit of max dead band"
 generic   minhi     S_P   1              "hi limit of min dead band"
 generic   minlo     S_P   0.9            "lo limit of min dead band"

PARAMETER_PROCESSING

/* Check that bands don't overlap */

   IF maxhi <= maxlo
   OR maxlo <= minhi
   OR minhi <= minlo THEN
     CALL nmf_error ("Parameters must be ordered
                      minlo<minhi<maxlo<maxhi")
   END_IF


END_MODEL
```

Example 3

```
CONTINUOUS_MODEL  bdzont

ABSTRACT
  "A dynamic zone model for air-exchange modelling. Bidirectional
   transports of energy plus a mass fraction are modelled.

   Alternative form of BDZONE used for demand controlled
   ventilation.

   Compared to BDZONE this model has two extra interfaces
   for control sensors."

EQUATIONS

/* fraction conservation */

   0 = - X'* xCap *1. / 3600
       + Occ * xfRate
       + Xf_source + Xf_0 + SUM i3=1, n Xf[i3] END_SUM
   GOOD_INVERSES (X) ;

/* mass conservation */

   0 = M_0 + SUM i=1, n M[i] END_SUM
   BAD_INVERSES () ;

/* energy conservation */

   0 = Q_zone + Q_0 + SUM i2=1, n Q[i2] END_SUM
   BAD_INVERSES () ;

LINKS

/* type         name           variables... */

   BidirX       terminal_0     P, POS_IN M_0, T_zone, POS_IN Q_0,
                               X, POS_IN Xf_0;

   FOR i = 1, n
     BidirX     terminal[i]    P, POS_IN M[i], T_zone, POS_IN Q[i],
                               X, POS_IN Xf[i];
   END_FOR;

   Tq           heat_load      T_zone, POS_IN Q_zone;
   T            air_temp       T_zone;
   X_y          fract          X;
   Z            people         Occ;
   X_h          source         POS_IN Xf_source;
```

```
VARIABLES

/* type         name       role  def min max  description */

  Fraction_y    X          OUT                "zone fraction"
  MassFlow_h    M_0        OUT                "terminal 0 massflow"
  MassFlow_h    M[n]       IN                 "terminal i massflow"
  Pressure      P          IN                 "zone floor level press"
  HeatFlux      Q_0        OUT                "terminal 0 HeatFlux"
  HeatFlux      Q[n]       IN                 "terminal i HeatFlux"
  Temp          T_zone     IN                 "zone temperature"
  FractFlow_h   Xf_0       IN                 "terminal 0 transport"
  FractFlow_h   Xf[n]      IN                 "terminal i transport"
  HeatFlux      Q_zone     IN                 "heat gain/loss in zone"
  FractFlow_h   Xf_source  IN                 "Mass fraction source
                                               (or sink)"
  Factor        Occ        IN                 "No of occupants"

MODEL_PARAMETERS

/* type  name  role  def min max     description */
   INT   n     SMP   1   0   BIGINT  "Number of links minus one"

PARAMETERS

/* type        name     role  def  min   max  description */
  Length       za       S_P   0    -BIG  BIG  "zone floor height
                                               relative to ground"
  Length       h        S_P   2.4  SMALL BIG  "zone height"
  Area         a        S_P   10   SMALL BIG  "zone floor area"
  FractFlow_h  xfRate   S_P   10   0     BIG  "contaminant src/person"
  Density      rho_20   S_P   1.2  SMALL 1.3  "density at ground press"
  mass         xCap     C_P                   "zone capacity of X [kg]"

PARAMETER_PROCESSING

  xCap := a * h * rho_20 ;


END_MODEL
```

Example 4

```
CONTINUOUS_MODEL utleak

ABSTRACT
   "A powerlaw leak model between zone and environment.
   Bidirectional transports of mass, energy and a mass fraction.
   A (building) Face parameter gives reference to specific set of
   global environment data. face=0 gives precedence to
   locally selected environment data."

EQUATIONS

/* driving pressure difference*/

Rho_in := rho_20*(293/(T_in + 273));
Rho_out := rho_20*(293/(T_out + 273));

Dp := P_in - Rho_in*G*zr_in - (P_out - Rho_out*G*zr_out)-Rho*G*dz;

/* powerlaw massflow equation */
   M / 3600 = IF LINEARIZE (1) THEN c * Dp
              ELSE_IF abs (Dp) < dp0 THEN c0 * Dp
              ELSE_IF Dp > 0 THEN c * Dp**n
              ELSE -c * (-Dp)**n
              END_IF
   BAD_INVERSES () ;

/* convected heat through leak */
   Q = IF LINEARIZE (1) THEN (T_in - T_out) / 2
       ELSE_IF M > 0.0 THEN cp * T_in * M/3600
       ELSE cp * T_out * M/3600
       END_IF
   GOOD_INVERSES (Q) ;

/* fraction transported through leak*/
   Xf = IF LINEARIZE (1) THEN (X_in - X_out) / 2
        ELSE_IF M > 0.0 THEN X_in * M/3600
        ELSE X_out * M/3600
        END_IF
   GOOD_INVERSES (Xf);

/* density of crack air is an assigned state (due to hysteresis) */
   Rho := IF LINEARIZE (1) THEN rho_20
          ELSE rho_20*(293/(IF EVENT(Mm, M) > 0 THEN T_in
                            ELSE T_out
                            END_IF + 273))
          END_IF ;

LINKS

/* type      name     variables... */

   BidirX    inside   P_in, POS_IN  M, T_in, POS_IN  Q, X_in,
                      POS_IN  Xf;
   BidirX    outside  P_out, POS_OUT M, T_out,POS_OUT Q,X_out,
                      POS_OUT Xf;
```

```
VARIABLES

/* type      name     role def  min      max  description */

Density     Rho      A_S  1.2  .5        3    "density of leak air"
Density     Rho_in   LOC  1.2  .5        3    "density of zone air"
Density     Rho_out  LOC  1.2  .5        3    "density of outside air"
MassFlow_h  M        OUT  0    -BIG      BIG  "massflow through leak"
Pressure    P_in     IN   1    -BIG      BIG  "inside floor pressure"
Pressure    P_out    IN   2    -BIG      BIG  "outside grnd pressure"
Temp        T_in     IN   20   ABS_ZERO BIG  "inside temperature"
Temp        T_out    IN   20   ABS_ZERO BIG  "outside temperature"
HeatFlux    Q        OUT  0    -BIG      BIG  "heat moved by massflow"
Fraction_y  X_in     IN   .1   0         1    "zone fraction"
Fraction_y  X_out    IN   .1   0         1    "environment fraction"
FractFlow_h Xf       OUT  0    -BIG      BIG  "fraction moved"
MassFlow_h  Mm       A_S                      "massflow memory"
Pressure    Dp       LOC                      "effective press diff"


PARAMETERS

/* type    name    role def  min   max  description */

/* easy access parameters */
/* priority order: ela, c_t */

Generic    c_t     S_P  1    0     BIG  "powerlaw coeff [kg/(s Pa**n)]"
Generic    n       S_P  .5   .5    1.0  "powerlaw exponent [dimless]"
Area       ela     S_P  0    0     BIG  "equivalent leakage area
                                         at Dp=4 Pa (C_d = 1)"
Factor     face    S_P  0    0     BIG  "Building Face number,
                                         face=0 => local data is used"
length     dz      S_P  0    -BIG  BIG  "rise fr in to out (may be <0)"
length     zr_in   S_P  0    0     BIG  "leak height from floor zone"
Length     za_in   S_P  0    -BIG  BIG  "absolute floor level of zone"
HeatCapM   cp      S_P  1006 .5E3  3E3  "air cp"
Pressure   dp0     S_P  .1   SMALL BIG  "limit for linear flow"
Density    rho_20  S_P  1.2  SMALL 1.3  "density at ground pressure"

/* derived parameters
c is derived if ela > 0 */

Generic    c       C_P                  "powerlaw coefficient
                                         [kg/(s Pa**n)]"
Generic    c0      C_P                  "linear coefficient"
Length     zr_out  C_P                  "leak height from ground level"

PARAMETER_PROCESSING

   zr_out := za_in + zr_in + dz;

   c := IF ela > SMALL THEN
           ela * SQRT(4 * 2 * rho_20)/4**n
        ELSE
           c_t
        END_IF ;

   c0 := c * dp0**(n-1) ;

END_MODEL
```

Example 5

```
CONTINUOUS_MODEL VxFan

ABSTRACT "Fan with splined fan curve.
          Volume flow in fan curve is converted initially
          to mass flow."

EQUATIONS

/* calculate Dp as a spline function of mass flow M */
   CALL splint (mf_nom,dp_nom,y2,n,M,Dp_spl,Slope_spl) ;

   Dp := IF LINEARIZE (1) THEN
           dp_nom[1] + (M - mf_nom[1]) * slope
         ELSE Dp_spl
         END_IF ;

/* massflow equation */
   0 = -P2 + P1 + Dp
   BAD_INVERSES () ;

LINKS

/* type         name         variables... */

   VentX        inlet        P1, POS_IN M, T, X ;
   VentX        outlet       P2, POS_OUT M, T, X ;


VARIABLES

/* type         name         role def  min      max  description */

  MassFlow_h   M            IN   1.   -BIG     BIG  "massflow thru fan"
  Pressure     P1           IN   -10. SMALL    BIG  "terminal 1 press"
  Pressure     P2           OUT  0    SMALL    BIG  "terminal 2 press"
  Temp         T            IN   15.  ABS_ZERO BIG  "temperature"
  Fraction_y   X            IN   .1   0.       BIG  "pollutant fraction"
  Pressure     Dp           LOC                     "eff pressure diff"
  Pressure     Dp_spl       LOC                     "Dp_spl = spline(M)"
  Generic      Slope_spl    LOC                     "local slope of
                                                     spline"

MODEL_PARAMETERS

/* type         name         role def  min  max  description */
   INT          n            SMP  3    2    10   "nr of pts on fan curve"
```

```
PARAMETERS

/* type         name      role def min    max  description */

/* easy access parameters */
/* fan curve */
   Pressure     dp_nom[n] S_P  0.  SMALL  BIG  "pressure rise, nominal"
   VolFlow_h    vf_nom[n] S_P  0.  SMALL  BIG  "volume flow"

/* derived parameters */
   Generic      slope     C_P                  "mean slope of fan curve"
   MassFlow_h   mf_nom[n] C_P                  "mass flow for fan curve"
   Generic      y2[n]     C_P                  "2nd deriv for spline"

PARAMETER_PROCESSING

/* convert fan curve to mass flow */
   FOR i = 1, n
     mf_nom[i] := vf_nom[i] * RHO_AIR ;
   END_FOR ;

   slope := (dp_nom[n] - dp_nom[1]) / (mf_nom[n] - mf_nom[1]) ;

/* prepare spline curve */
   y2[1] := 0 ;
   CALL spline (mf_nom, dp_nom, n, 1E30, 1E30, y2) ;

FUNCTION VOID SPLINT(XA,YA,Y2A,N,X,Y,YPRIM)

/* Calculate Y(X) from spline*/

LANGUAGE F77

INPUT
  INT N;
  FLOAT XA[N],YA[N],Y2A[N],X,Y,YPRIM;

CODE
  SUBROUTINE SPLINT(XA,YA,Y2A,N,X,Y,YPRIM)

  INTEGER N
      REAL XA(N),YA(N),Y2A(N),X,Y,YPRIM

* Calculate Y(X) from spline XA,YA,Y2A
* XA(N),YA(N) = pivoting points,
* Y2A(N) = 2nd deriviatives in points.

  INTEGER K,KLO,KHI
      REAL A,B,H

      KLO = 1
      KHI = N
1     IF (KHI-KLO.GT.1) THEN
         K = (KHI+KLO)/2
         IF (XA(K).GT.X)THEN
            KHI = K
         ELSE
            KLO = K
         ENDIF
         GOTO 1
      ENDIF

      H = XA(KHI) - XA(KLO)
      IF (H.LE.0.) THEN
         PRINT *, 'Arguments to SPLINE not ascending'
```

```
            STOP
         ENDIF

         A = (XA(KHI) - X) / H
         B = (X-XA(KLO)) / H
         Y = A*YA(KLO)+B*YA(KHI) +
     &         ((A**3-A)*Y2A(KLO)+(B**3-B)*Y2A(KHI))*(H**2) / 6.
         YPRIM = (YA(KHI)-YA(KLO) +
     &         (Y2A(KHI)*(3*B**2-1.) - Y2A(KLO)*(3*A**2-1.)) / 6.
     &          ) / H
         RETURN
         END
END_CODE

END_FUNCTION

FUNCTION VOID SPLINE(X,Y,N,YP1,YPN,Y2)

/* Prepare SPLINE parameters for SPLINT */

LANGUAGE F77

INPUT
   INT N;
   FLOAT X[N],Y[N],YP1,YPN,Y2[N];

CODE

   SUBROUTINE SPLINE(X,Y,N,YP1,YPN,Y2)
         INTEGER NMAX
         PARAMETER (NMAX = 100)
         INTEGER N
         REAL X(N),Y(N),YP1,YPN,Y2(N),U(NMAX)

* Prepare SPLINE parameters for SPLINT
* X(N),Y(N) = pivoting points,
* YP1,YPN = 1st deriviatives in end points,
* Y2(N) = calculated 2nd derivatives.


         INTEGER I,K
         REAL P,QN,SIG,UN

* Check increasing Xs
         DO 10 I = 1,N-1
            IF (X(I+1) - X(I) .LE. 0.) THEN
               PRINT *,'Arguments to SPLINT not ascending'
               STOP
            ENDIF
    10   CONTINUE

         IF (YP1 .GT. .99E30) THEN
            Y2(1) = 0.
            U(1) = 0.
         ELSE
            Y2(1) = -0.5
            U(1) = (3./(X(2)-X(1)))*((Y(2)-Y(1))/(X(2)-X(1))-YP1)
         ENDIF

         DO 11 I = 2,N-1
            SIG = (X(I)-X(I-1)) / (X(I+1)-X(I-1))
            P = SIG*Y2(I-1)+2.
            Y2(I) = (SIG-1.) / P
            U(I) = (6.*((Y(I+1)-Y(I))/(X(I+1)-X(I))-(Y(I)-Y(I-1))
     &         / (X(I)-X(I-1)))/(X(I+1)-X(I-1))-SIG*U(I-1))/P
```

```
11    CONTINUE

      IF (YPN.GT..99E30) THEN
         QN = 0.
         UN = 0.
      ELSE
         QN = 0.5
         UN = (3./(X(N)-X(N-1)))*(YPN-(Y(N)-Y(N-1))/(X(N)-X(N-1)))
      ENDIF
      Y2(N) = (UN-QN*U(N-1)) / (QN*Y2(N-1)+1.)

      DO 12 K = N-1,1,-1
         Y2(K) = Y2(K)*Y2(K+1) + U(K)
12    CONTINUE

      RETURN
      END
END_CODE

END_FUNCTION

END_MODEL
```

Example 6

```
CONTINUOUS_MODEL VxExhT

ABSTRACT "Exhaust Terminal.
          Linear flow below limit 'dp0', and if LINEARIZE."

EQUATIONS

   Rho := rho_20 * (20. - ABS_ZERO) / (T - ABS_ZERO) ;
   Dp := P1 - P2 - zr1 * G * Rho ;

/* powerlaw massflow equation */
   0 = -M / 3600 + IF LINEARIZE (1) THEN c_turb * Dp
                   ELSE_IF Dp < dp0 THEN c_lin * Dp
                   ELSE c_turb * sqrt (Dp)
                   END_IF
   BAD_INVERSES ();

/* convected heat through terminal */
   Q = IF LINEARIZE (1) THEN T
       ELSE cp * T * M / 3600
       END_IF
   GOOD_INVERSES (Q);

/* fraction transported through terminal */
   Xf = IF LINEARIZE (1) THEN X
        ELSE X * M / 3600
        END_IF
   GOOD_INVERSES (Xf);

LINKS

/* type      name      variables... */

   BidirX    zone      P1, POS_IN M, T, POS_IN Q, X, POS_IN Xf ;
   VentX     outlet    P2, POS_OUT M, T, X ;

VARIABLES

/* type         name role def min      max   description */

  massflow_h    M    OUT  0.  -BIG     BIG   "massflow through leak"
  Pressure      P1   IN   2.  SMALL    BIG   "terminal 1 pressure"
  Pressure      P2   IN   1.  SMALL    BIG   "terminal 2 pressure"
  temp          T    IN   15. ABS_ZERO BIG   "temperature"
  HeatFlux      Q    OUT  0.  -BIG     BIG   "heat convected by
                                              massflow"
  fraction_y    X    IN   .1  0.       BIG   "pollutant fraction"
  FractFlow_h   Xf   OUT  0.  -BIG     BIG   "pollution transport"
  Pressure      Dp   LOC                     "eff pressure diff"
  density       Rho  LOC                     "air density"
```

```
PARAMETERS

/* type      name    role def  min   max  description */

/* easy access parameters */
/* priority order: c_t, xi */

  generic    c_t     S_P  0    0     BIG  "powerlaw coefficient"
  length     d       S_P  .25  SMALL BIG  "inner diameter"
  generic    xi      S_P  10.  SMALL BIG  "loss coeff"
  length     zr1     S_P  0    0     BIG  "leak height from floor"
  HeatCapM   cp      S_P  1006 500   3000 "air cp"
  Pressure   dp0     S_P  .1   SMALL BIG  "limit for linear flow"
  Density    rho_20  S_P  1.2  SMALL BIG  "reference density"
/* derived parameters */
  area       a       C_P                  "cross section area"
  generic    c_lin   C_P                  "laminar coefficient"
  generic    c_turb  C_P                  "flow characteristic"

PARAMETER_PROCESSING

   a := PI * d * d / 4. ;

/* Check alternative definitions of c_turb */

   c_turb := IF c_t != 0. THEN
                c_t
             ELSE_IF xi == 0. THEN
                0.
             ELSE
                a * (2. * rho_20 / xi)**0.5
             END_IF ;

   IF c_turb == 0 THEN
      CALL nmf_error
        ("parameters missing (c_t or xi) for exhaust terminal ")
   END_IF ;

   c_lin := c_turb / sqrt (dp0) ;


END_MODEL
```

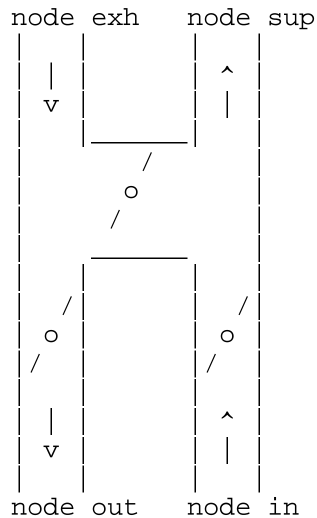
Example 7

```
CONTINUOUS_MODEL VxSupT

ABSTRACT "Supply Terminal.
          Linear flow below limit 'dp0', and if LIN."

EQUATIONS

   Rho := rho_20 * (20. - ABS_ZERO) / (T2 - ABS_ZERO) ;
   Dp := P1 - P2 + zr2 * G * Rho ;

/* powerlaw massflow equation */
   0 = -M / 3600 + IF LINEARIZE (1) THEN c_turb * Dp
                   ELSE_IF Dp < dp0 THEN c_lin * Dp
                   ELSE c_turb * sqrt (Dp)
                   END_IF
   BAD_INVERSES ();

/* convected heat through terminal */
   Q = IF LINEARIZE (1) THEN T1
       ELSE cp * T1 * M / 3600
       END_IF
   GOOD_INVERSES (Q);

/* fraction transported through terminal */
   Xf = IF LINEARIZE (1) THEN X1
        ELSE X1 * M / 3600
        END_IF
   GOOD_INVERSES (Xf);

LINKS

/* type         name        variables... */

   VentX        inlet       P1, POS_IN M, T1, X1 ;
   BidirX       zone        P2, POS_OUT M, T2, POS_OUT Q,
                            X2, POS_OUT Xf ;

VARIABLES

/* type         name  role def min      max   description */

  massflow_h    M     OUT  0.  -BIG     BIG   "massflow through leak"
  Pressure      P1    IN   2.  SMALL    BIG   "pressure in"
  Pressure      P2    IN   1.  SMALL    BIG   "pressure out"
  temp          T1    IN   15. ABS_ZERO BIG   "temperature in"
  temp          T2    IN   15. ABS_ZERO BIG   "temperature zone"
  HeatFlux      Q     OUT  0.  -BIG     BIG   "heat convected by
                                                massflow"
  fraction_y    X1    IN   .1  0.       BIG   "pollutant fraction in"
  fraction_y    X2    IN   .1  0.       BIG   "pollutant fraction
                                                zone"
  FractFlow_h   Xf    OUT  0.  -BIG     BIG   "pollution transport"
  Pressure      Dp    LOC                     "eff pressure diff"
  density       Rho   LOC                     "air density"
```

```
PARAMETERS

/* type      name     role def  min   max  description */
/* easy access parameters */
/* priority order: c_t, xi */

  generic    c_t      S_P  0    0     BIG  "powerlaw coefficient"
  length     d        S_P  .25  SMALL BIG  "inner diameter"
  generic    xi       S_P  10.  0     BIG  "loss coeff"
  length     zr2      S_P  0    0     BIG  "leak height from floor"
  HeatCapM   cp       S_P  1006 500   3000 "air cp"
  Pressure   dp0      S_P  .1   SMALL BIG  "limit for linear flow"
  Density    rho_20   S_P  1.2  SMALL BIG  "reference density"

/* derived parameters */
  area       a        C_P                  "cross section area"
  generic    c_lin    C_P                  "laminar coefficient"
  generic    c_turb   C_P                  "flow characteristic"

PARAMETER_PROCESSING

   a := PI * d * d / 4. ;

/* Check alternative definitions of c_turb */

   c_turb := IF c_t != 0. THEN
                c_t
             ELSE_IF xi == 0. THEN
                0.
             ELSE
                a * (2. * rho_20 / xi)**0.5
             END_IF ;

   IF c_turb == 0 THEN
      CALL nmf_error
        ("parameters missing (c_t or xi) for supply terminal ")
   END_IF ;

   c_lin := c_turb / sqrt (dp0) ;


END_MODEL
```

Example 8

```
CONTINUOUS_MODEL VxMix

ABSTRACT "Mixing box. Three dampers move in parallel.
          Incoming control signal (0-1) defines required fraction
          of outside air.
          Resistances are linear functions of damper 'position'."

/*
```



```
*/

EQUATIONS

/* pressure differences */
   DpOut := PExh - POut ;
   DpIn := PIn - PSup ;
   DpRet := PExh - PSup ;

/* flow characteristics as functions of position */
   COut = IF LINEARIZE (1) THEN
             (cOutMin + cOutMax) / 2
          ELSE
             cOutMin * (1 - Position) + cOutMax * Position
          END_IF
   GOOD_INVERSES(COut);

   CRet = IF LINEARIZE (1) THEN
             (cRetMin + cRetMax) / 2
          ELSE
             cRetMax * (1 - Position) + cRetMin * Position
          END_IF
   GOOD_INVERSES (CRet);

   COutLin := COut / sqrt (dp0) ;
   CRetLin := CRet / sqrt (dp0) ;

/* mass conservation */
   MRet := MExh - MOut ;

   0 = MExh + MIn - MSup - Mout
   BAD_INVERSES();
```

```
/* desired fresh air fraction */
   0 = - InSignal + IF LINEARIZE(1) THEN
                       Position
                    ELSE
                       MIn / MSup
                    END_IF
   GOOD_INVERSES (Insignal) ;

/* powerlaw massflow equations*/

   0 = - MOut / 3600 + IF LINEARIZE (1) THEN cOut * DpOut
                          ELSE_IF DpOut < dp0 THEN cOutLin * DpOut
                          ELSE COut * sqrt (DpOut)
                          END_IF
   GOOD_INVERSES (MOut) ;

   0 = -MIn / 3600 + IF LINEARIZE (1) THEN cOut * DpIn
                        ELSE_IF DpIn < dp0 THEN cOutLin * DpIn
                        ELSE COut * sqrt (DpIn)
                        END_IF
   GOOD_INVERSES (MIn);

   0 = -MRet / 3600 + IF LINEARIZE (1) THEN cRet * DpRet
                         ELSE_IF DpRet < dp0 THEN cRetLin * DpRet
                         ELSE CRet * sqrt (DpRet)
                         END_IF
   GOOD_INVERSES (MRet);

/* temperature mix */
   0 = - TSup + IF LINEARIZE (1) THEN TIn
                ELSE TExh * (1. - InSignal) + TIn * InSignal
                END_IF
   GOOD_INVERSES(TSup);

/* fraction mix */
   0 = - XSup + IF LINEARIZE (1) THEN XIn
                ELSE XExh * (1. - InSignal) + XIn * InSignal
                END_IF
   GOOD_INVERSES(XSup);

LINKS

  /* type      name        variables... */

  VentX        exhaust     PExh, POS_IN MExh, TExh, XExh ;
  VentX        supply      PSup, POS_OUT MSup, TSup, XSup ;
  VentX        outlet      POut, POS_OUT MOut, TExh, XExh ;
  VentX        intake      PIn, POS_IN MIn, TIn, XIn ;

  ControlLink  freshair    InSignal ;
```

```
VARIABLES

/* type        name      role def   min      max    description */

  control      InSignal IN   1.    0.       1.    "desired fresh air
                                                   fraction"

  massflow_h  MExh      OUT  1.    -BIG     BIG   "exhaust mass flow"
  massflow_h  MSup      OUT  1.    -BIG     BIG   "supply mass flow"
  massflow_h  MOut      OUT  1.    -BIG     BIG   "outlet mass flow"
  massflow_h  MIn       OUT  1.    -BIG     BIG   "intake mass flow"
  massflow_h  MRet      LOC                       "return mass flow"

  pressure    PExh      IN   3.    SMALL    BIG   "exhaust press"
  pressure    PSup      IN   0.    SMALL    BIG   "supply pressure"
  pressure    POut      IN   2.    SMALL    BIG   "outlet pressure"
  pressure    PIn       IN   1.    SMALL    BIG   "intake pressure"

  temp        TExh      IN   15.   ABS_ZERO BIG   "exhaust temperature"
  temp        TSup      OUT  15.   ABS_ZERO BIG   "supply temperature"
  temp        TIn       IN   15.   ABS_ZERO BIG   "intake temperature"

  fraction_y  XExh      IN   1000. 0.       BIG   "exhaust fraction"
  fraction_y  XSup      OUT  1000. 0.       BIG   "supply fraction"
  fraction_y  XIn       IN   1000. 0.       BIG   "intake fraction"

  generic     COut      OUT                       "flow char in/out"
  generic     CRet      OUT                       "flow char return"

  pressure    DpOut     LOC                       "press diff out"
  pressure    DpIn      LOC                       "press diff in"
  pressure    DpRet     LOC                       "press diff return"


  factor      Position OUT                        "damper position"
  generic     cOutLin  LOC                        "laminar coeff in/out"
  generic     cRetLin  LOC                        "laminar coeff return"

PARAMETERS

/* type       name     role def  min   max  description */


/* easy access parameters */
  generic    cOutMin  S_P  0.5  0.    BIG  "flow coeff min in/out"
  generic    cOutMax  S_P  1.   SMALL BIG  "flow coeff max in/out"
  generic    cRetMin  S_P  0.5  0.    BIG  "flow coeff min return"
  generic    cRetMax  S_P  1.   SMALL BIG  "flow coeff max return"

  Pressure   dp0      S_P  .1   SMALL BIG  "limit for linear flow"

END_MODEL
```