Towards HYSDEL 3.0.alpha

Martin Herceg and Michal Kvasnica

Contents

1 HYSDEL 2.0.5 – Main Features **1**
1.1 Mixed-Logical Dynamic System Description 1
1.2 HYSDEL Language 1
1.3 HYSDEL Compiler 2
1.4 Beta Version of HYSDEL 2
1.5 Implementation of Vector HYSDEL 3

2 HYSDEL 3.0.alpha – Features **4**
2.1 Extended Description of MLD system 4
2.2 HYSDEL Language 6
2.2.1 Preliminaries 6
2.2.2 List of Language Changes 7
2.2.3 INTERFACE Section 7
2.2.4 IMPLEMENTATION Section 16
2.3 Merging of HYSDEL Files 31
2.4 Implementation of new HYSDEL 34
2.5 EXAMPLES 35

1 HYSDEL 2.0.5 – Main Features

1.1 Mixed-Logical Dynamic System Description

Mixed Logical Dynamical (MLD) systems describe in general the behavior of linear discrete-time systems with integrated logical rules. The basic principle, how logical rules are incorporated into overall description, is in transformation to a set of linear inequalities as shown in [1]. HYSDEL inherently considers this description as fundamental for further computations, which takes the form of

$$\begin{aligned} x(k+1) &= Ax(k) + B_1 u(k) + B_2 \delta(k) + B_3 z(k) && \text{(1.1a)} \\ y(k) &= Cx(k) + D_1 u(k) + D_2 w(k) + D_3 z(k) && \text{(1.1b)} \\ & E_2 \delta(k) + E_3 z(k) \leq E_1 u(k) + E_4 x(k) + E_5 && \text{(1.1c)} \end{aligned}$$

where $x \in \mathbb{R}^{n_{xr}} \times \{0,1\}^{n_{xb}}$ is a vector of continuous and binary states, $u \in \mathbb{R}^{n_{ur}} \times \{0,1\}^{n_{ub}}$ are the inputs, $y \in \mathbb{R}^{n_{yr}} \times \{0,1\}^{n_{yb}}$ vector of outputs, $\delta \in \{0,1\}^{n_d}$ represent auxiliary binary, $z \in \mathbb{R}^{n_z}$ continuous variables, respectively, and A, B_1, B_2, B_3, C, D_1, D_2, D_3, E_2, E_3, E_1, E_4, E_5 are matrices of suitable dimensions. For a give state $x(k)$ and input $u(k)$ the evolution of the MLD system (1.1) is determined by solving $\delta(k)$ and $z(k)$ from (1.1c) and updating $x(k+1)$, $y(k)$.

However, the interpretation of MLD system (1.1) was sometimes misleading when formulating control problems and several shortcomings have been identified. Precisely, the form (1.1c) does not explicitly cover equality constraints and they are treated as double sided inequalities. Additionally, binary and real variables are treated not equally, i.e. sometimes they are part of one vector, and sometimes they are separated. Another identified shortcoming was that the current MLD description does not consider affine terms directly which are are important for modeling.

1.2 HYSDEL Language

HYSDEL language enables to transform information about the modeled system to a HYSDEL code. Once the HYSDEL code is available, it is then further processed for other purposes like process simulation. Current HYSDEL language has several specifications, which are required when writing HYSDEL code. For instance each new declared variable has to be a scalar. This is not a problem for writing a small HYSDEL code, but it becomes apparent for larger systems. Similarly, the length of the code grows rapidly, when one has to write equations row-wise and the consequent code revision is not an easy task. Example of a standard HYSDEL language is given next:

Example 1 Standard HYSDEL language requires all variables to be scalars.

```
STATE {
 REAL x1, x2;
 }
 INPUT {
```

```
  REAL u1, u2;
}
PARAMETER {
  REAL a11, a12, a21, a22;
}
CONTINUOUS {
  x1=a11*x1 + a12*x2 + u1;
  x2=a21*x1 + a22*x2 + u2;
}
```

Note that each equation is written element-wise, which results in many lines of HYSDEL source code.

1.3 HYSDEL Compiler

Current way of obtaining an executable code of MLD model has to come trough a HYSDEL compiler, as shown in Fig. 1.1.

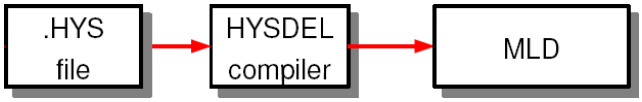

Figure 1.1: Current path of generating executable MLD employs the HYSDEL compiler.

1.4 Beta Version of HYSDEL

Extension of basic HYSDEL syntax to the beta version removed in fact main disadvantages of HYSDEL. Beta version is not officially released to public and it is under development. Main features covering the beta version are as follows:

- support of vectorized variables, as well as matrices
- provides indexed access to vectors/matrices
- contains iteration loops of type FOR and nested FOR loops
- provides simple model refinement, i.e. model optimization
- still uses the HYSDEL compiler to get MLD

Since the vectors and matrices are supported by this version, the beta version will be referred to as Vector HYSDEL in the sequel.

Here is a simple example of Vector HYSDEL, which corresponds to example 1.

Example 2 Vector HYSDEL allows to use vectorized code.

```
STATE {
  REAL X(2);
}
INPUT {
  REAL U(2);
```

```
}
PARAMETER {
  REAL A=[0.1,1; 0, 0.5];
}
CONTINUOUS {
  X = A*X + U;
}
```

Although the size of HYSDEL code is reduced, the operation involved in processing such file still remains complete, as will be shown in the sequel.

1.5 Implementation of Vector HYSDEL

In the Vector HYSDEL implementation phase the process takes several intermediate steps, as illustrated in Fig. 1.2. Although the vectorized description is easy to use, in the consequent steps it is required to obtain a de-vectorized description in the original HYSDEL syntax and rely on current MLD compiler. Basically, the extended HYSDEL language is an overloaded feature and needs to be decomposed to a classic (non-vectorized) HYSDEL language. This is achieved via parsing through XML toolbox and obtaining a non-vectorized HYSDEL code. Disadvantages of this approach is that involved XML parser consumes a lot of parsing time (typically more than 20 seconds) and further model optimization is difficult to handle. Furthermore, it is not straightforward to generally adjust the procedure for different operating systems.

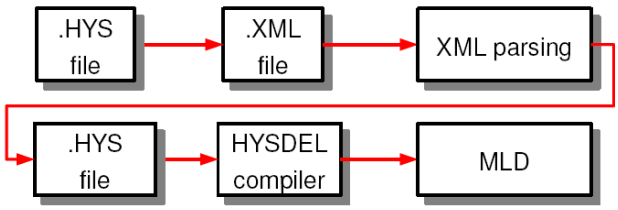

Figure 1.2: Current path of generating executable MLD models comprises of numerous intermediate-steps.

2 HYSDEL 3.0.alpha – Features

This chapter describes new features available in the version 3.0.alpha of the HYSDEL. The aim will be to focus on differences comparing to HYSDEL 2.0.5 version and interpret the new syntax on simple examples.

2.1 Extended Description of MLD system

To overcome identified shortcomings of the description (1.1) the model is rewritten into more convenient form. Arising from requirement, that each variables should be clearly distinct, new notations are adopted. More precisely, the new MLD description is given by

$$\begin{aligned} x(k+1) &= Ax(k) + B_u u(k) + B_{aux} w(k) + B_{aff} && \text{(2.1a)} \\ y(k) &= Cx(k) + D_u u(k) + D_{aux} w(k) + D_{aff} && \text{(2.1b)} \\ & E_x x(k) + E_u u(k) + E_{aux} w(k) \ \leq \ E_{aff} && \text{(2.1c)} \\ \{\text{indices}\} & J_x,\ J_u,\ J_w,\ J_{eq} && \text{(2.1d)} \end{aligned}$$

where the auxiliary vector $w(k)$ comprises of two elements $w(k) = [\delta(k),\ z(k)]^T$. Comparing to previous description (1.1), these changes are visible

- matrices B_1, D_1, and E_1 are referred to as B_u, D_u, and E_u, respectively
- auxiliary variables $\delta(k)$ and $z(k)$ are merged together in one vector called $w(k) = [\delta(k),\ z(k)]^T$
- matrix couples B_2 - B_3, D_2 - D_3, E_2 - E_3 are referred to as $B_{aux} = [B_2\ B_3]$, $D_{aux} = [D_2\ D_3]$, $E_{aux} = [E_2\ E_3]$, respectively
- affine terms B_{aff}, D_{aff} are included
- E_4 and E_5 are now referred to as E_x and E_{aff}, respectively. Moreover, since inequality (2.1c) is written in a standard form, i.e. $Av \leq b$, and comparing to (1.1), their relations are $E_x = -E_4$ and $E_u = -E_1$.
- indices (2.1d) indicate which variables are real, and which are binary for states J_x, inputs J_u, auxiliary variables J_w and additionally, J_{eq} corresponds to indices with equality constraints

Structure of the indexed sets $J_x,\ J_u,\ J_w$ is column-wise, distinguishing between binary and real variables with strings. Precisely, string 'r' refers to real and string 'b' refers to binary/Boolean variable, e.g.

$$\{J_x,\ J_u,\ J_w\} = \begin{pmatrix} \texttt{'r'} \\ \texttt{'r'} \\ \texttt{'b'} \\ \texttt{'r'} \end{pmatrix} \quad \text{corresponds to} \quad \begin{pmatrix} \texttt{REAL} \\ \texttt{REAL} \\ \texttt{BOOL} \\ \texttt{REAL} \end{pmatrix}$$

To determine location of equality constraints, the vector of indices J_{eq} will refer to their corresponding rows in (2.1c), e.g.

$$J_{eq} = \begin{pmatrix} 1 \\ 5 \\ 8 \end{pmatrix} \quad \text{corresponds to} \quad \begin{pmatrix} \text{1st row is an equality constraint} \\ \text{5th row is an equality constraint} \\ \text{8th row is an equality constraint} \end{pmatrix}$$

[Although labeling by strings clearly helps to distinguish the type of variables, it is not suitable for further operations based on indexing. For this purpose, the information about the position of variables in a vector will be stored in a substructure '`j`'. Notation remains the same as in 2.0.5 version, and internal access is Matlab-like, e.g.]

```
S.j.xr    /* indices of REAL states */
 .j.xb    /* indices of BOOL states */
 .j.ur    /* indices of REAL inputs */
 .j.ub    /* indices of BOOL inputs */
 .j.yr    /* indices of REAL outputs */
 .j.yb    /* indices of BOOL outputs */
 .j.d     /* indices of BOOL auxiliary variables */
 .j.z     /* indices of REAL auxiliary variables /*
 .j.eq    /* indices of equality constraints */
 .j.ineq  /* indices of inequality constraints */
```

Remaining notations of HYSDEL 2.0.5 remain the same, e.g. dimension of the state `nx`, or binary state `nxr`, etc. Once the HYSDEL file proceeds through the compiler whole information will be stored in a structure format called `S`. Access to internal variables is Matlab-like, for instance one can access the dimension of states is as `S.nx`, and dimension of states of type real as `S.nxr`. Matrices introduced in MLD model (2.1) will be stored as

```
S.A
 .Bu
 .Baux
 .Baff
 .C
 .Du
 .Daux
 .Daff
 .Ex
 .Eu
 .Eaux
 .Eaff
```

Dimensions of the variables will have the following notations

```
S.nx  /* dimension of states */
 .nu  /* dimension of inputs */
 .ny  /* dimension of outputs */
 .nw  /* dimension of auxiliary variables */
 .nc  /* dimension of constraints (equalities + inequalities) */
```

and indexed set will be accessible through

```
S.J.X    /* string of indices ('r' or 'b') for states */
 .J.U    /* string of indices ('r' or 'b') for inputs */
 .J.Y    /* string of indices ('r' or 'b') for outputs */
 .J.W    /* string of indices ('r' or 'b') for auxiliary variables */
 .J.eq   /* numerical indices of equality constraints */
```

2.2 HYSDEL Language

This section points out the main differences comparing to old HYSDEL language. The structure follows the syntactical parts of HYSDEL scripting language. Before proceeding further, basics of the HYSDEL language will be given.

2.2.1 Preliminaries

A HYSDEL syntax was developed on C-language, and many symbols are similar. Structurally, a standard HYSDEL file comprises of two modes, namely INTERFACE and IMPLEMENTATION. Each of the sections is delimited by curly brackets e.g.

```
section {
        ...
}
```

and this holds for every kind of subsections as well. If one wants to add comments, this is simply done via C-like comments, i.e.

```
section {
        ...
/* this line is commented */
        ...
}
```

An example of the simplest HYSDEL file might look as follows

```
SYSTEM name {
/* example of HYSDEL file structure */
   INTERFACE {
      /* interface part serves as declaration of variables */
   }
   IMPLEMENTATION {
      /* implementation part defines relations between declared variables */
   }
}
```

Note that the file begins with SYSTEM string which is followed by the name. The name can be arbitrary, but it should be kept in mind that it usually points to a real object, therefore is recommended to use labels as e.g. tank, valve, pump, belt, etc. Other strings like INTERFACE and IMPLEMENTATION are obligatory. HYSDEL file can be created within any preferred text editor, however, the file should always have the suffix ".hys", e.g. `tank.hys`. More importantly, it is recommended to use the name of the SYSTEM also as the file name, e.g. system called `tank` will be a file `tank.hys` etc. This becomes reasonable if there are more HYSDEL files in one directory and helps to identify the files easily. In the next content, the aim is to interpret new syntactical changes comparing to HYSDEL 2.0.5 version.

2.2.2 List of Language Changes

The list of topics of syntactical changes in HYSDEL 3.0.alpha is briefly summarized in the sequel, while the details will be explained in particular sections.

- INTERFACE section
  - extensions of the INTERFACE section
  - declaration of INPUT, STATE, OUTPUT variables in scalar/vectorized form
  - declaration of parameters
  - declaration of subsystems
- IMPLEMENTATION section
  - extensions of the IMPLEMENTATION section
  - indexing
  - FOR and nested FOR loops
  - access to symbolic parameters
  - operators and built-in functions
  - AUX section
  - CONTINUOUS section
  - AUTOMATA section
  - LINEAR section
  - LOGIC section
  - AD section
  - DA section
  - MUST section
  - OUTPUT section
- Merging of HYSDEL files

2.2.3 INTERFACE Section

The INTERFACE section defines the main variables which appear for the given SYSTEM. More precisely, this section defines input, state and output variables distinguished by strings INPUT, STATE and OUTPUT. Moreover, additional variables which do not belong to these classes are supposed to be declared in the section called PARAMETER. A new feature of this version is that sometimes the block SYSTEM may contain other subsystems and this block should describe the overall behavior of the system. If this is the case, the MODULE section is to be present where subsystems are declared. This is the main change contrary to previous version.

Syntactical structure of the INTERFACE section has the following structure

```
INTERFACE { interface_item }
```

which comprises of curly brackets and *interface_item*. The *interface_item* may take only following forms

```
/* allowed INTERFACE items */
   MODULE { /* module_item */ }
   INPUT { /* input_item */ }
```

```
STATE { /* state_item */  }
OUTPUT { /* output_item */ }
PARAMETER { /* parameter_item */ }
```

where each item appears only once in this section. Each *interface_item* is separated at least with one space character and may be omitted, if it is not required. The order of each item can be arbitrary, it does not play a role for further processing.

Example 3 A structure of a standard HYSDEL file is shown here, where the INTERFACE items are separated by paragraphs, and curly brackets denote visible start- and end-points of each section.

```
SYSTEM name {
/* example of HYSDEL file */
   INTERFACE {
      /* declaration of variables, subsystems */
      MODULE {
         /* declaration subsystems */
         ...
      }
      INPUT {
         /* declaration input variables */
         ...
      }
      STATE {
         /* declaration state variables */
         ...
      }
      OUTPUT {
         /* declaration output variables */
         ...
      }
      PARAMETER {
         /* declaration of parameters */
         ...
      }
   }
   IMPLEMENTATION {
      /* relations between declared variables */
         ...
   }
}
```

Related subsections of the INTERFACE part will be explained in more detailed in the sequel.

INPUT section

In the input section are declared input variables of the SYSTEM which can be of type REAL (i.e. $u_r \in \mathbb{R}$), and BOOL (i.e. $u_b \in \{0, 1\}$). General syntax of the INPUT section remains the same, i.e.

```
INPUT { input_item }
```

where each new *input_item* is separated by at least one character space but the syntax of *input_item* differs for real and binary variables. The *input_item* for real variables takes the form of

```
REAL var [var_min, var_max];
```

where the string `REAL`, which denotes the type, is followed by `var` referring to name of the input variable, the strings `[var_min, var_max]` express the lower and upper bounds, and semicolon ";" denotes the end of the *input_item*. If there are more than one input variables, they are separated by commas ",", i.e.

```
REAL var1 [var_1_min, var_2_max], var2 [var_2min, var_2_max];
```

Example 4 Declaration of the scalar variable called "input_flow", which is bounded between 0 and 10 m^3/s will take the form

```
REAL input_flow [0, 10];
```

The *input_item* for Boolean variables takes the form of

```
BOOL var;
```

where the string `BOOL`, which denotes the type, is followed by `var` referring to name of the input variable, and semicolon ";" denotes the end of the *input_item*. Here the specification of bounds is not necessary because they are known but if even despite this one specifies the bounds on Boolean variables, HYSDEL will report an error. For more than one variables, separation by commas "," is required, i.e.

```
BOOL var1, var2, var3;
```

Example 5 Declaration of the scalar Boolean variables called "switch" and "running" will take the form

```
BOOL switch, running;
```

The main change comparing to previous version is that input variables may be defined as vectors, whereas the dimension of the input vector of real variables is n_{ur} and n_{ub} for binary variables. This allows to define vectorized variables in the meaning of $u_r \in \mathbb{R}^{n_{ur}}$, $u_b \in \{0,\ 1\}^{n_{ub}}$ and the corresponding syntax is as follows

```
REAL var(nur) [var_1_min, var_1_max;
               var_2_min, var_2_max;
                     ...,       ...;
               var_nur_min, var nur_max];
```

for variables of type REAL and

```
BOOL var(nub);
```

for Boolean variables. Note that according to expression in normal brackets "(" ")", which denotes the dimension of the vector, the lower and upper bounds are specified for each variable in this vector. These bounds are separated by commas "," and semicolon ";" denotes the end of row. If one declares variable in this way HYSDEL inherently assumes a column vector.

Note that the dimension of vector has to be always scalar and integer valued from the set $\mathbb{N}_+ = \{1,\ 2,\ \ldots\}$ and a particular value has to be always assigned. If the dimension of the vector is a symbolical parameter, HYSDEL will report an error. This holds similarly for STATE, OUTPUT and PARAMETER section.

Example 6 We want to declare the input real vector $u_r \in \mathbb{R}^3$ where the first variable may vary in $u_{r1} \in [-1,\ 2]$, the second variable in $u_{r2} \in [0.5, 1.3]$, and the third variable in $u_{r3} \in [-0.5, 0.5]$. Moreover, Boolean inputs of length 2, i.e. $u_b \in \{0,\ 1\}^2$ are present. The declaration of the INPUT section will take the form

```
INPUT {
  REAL ur(3) [-1, 2; 0.5, 1.3; -0.5, 0.5];
  BOOL ub(2);
}
```

where it is not required to separate the lower and upper bounds into rows of the HYSDEL code. Important is that the semicolon as separator is present.

STATE section

STATE section declares state variables of the SYSTEM which can be of type REAL (i.e. $x_r \in \mathbb{R}$), and BOOL (i.e. $x_b \in \{0,\ 1\}$) similarly as in the INPUT section. In general, the syntax of the STATE section is as follows

```
STATE { state_item }
```

where each new *state_item* is separated by at least one character space but the particular syntax of *state_item* differs for real and binary variables. The *state_item* for real variables takes the form of

```
REAL var [var_min, var_max];
```

where the string `REAL`, which denotes the type, is followed by `var` referring to name of the input variable, the strings `[var_min, var_max]` express the lower and upper bounds, and semicolon "`;`" denotes the end of the *state_item*. If there are more than one input variables, they are separated by commas ",", i.e.

```
REAL var1 [var_1_min, var_2_max], var2 [var_2_min, var_2_max];
```

Example 7 Declaration of the scalar variables called "position", which is bounded between 0 and 100 m, and "speed" (bounded from -10 to 10), will take the form

```
REAL position [0, 1000], speed [-10, 10];
```

The *state_item* for Boolean variables takes the form of

```
BOOL var;
```

where the string `BOOL`, which denotes the type, is followed by `var` referring to name of the Boolean state, and semicolon "`;`" denotes the end of the *state_item*. For more than one variables, separation by commas "," is required, i.e.

```
BOOL var1, var2, var3;
```

Example 8 Declaration of the scalar Boolean state called "is_open" will take the form

```
BOOL is_open;
```

The main syntax difference, comparing to previous version, is that both REAL and BOOL variables may be defined as vectors. Denoting the dimension of the real state vector as n_{xr} and dimension of Boolean state vector n_{xb} allows one to define vectors in the sense of $x_r \in \mathbb{R}^{n_{xr}}$, $x_b \in \{0,\ 1\}^{n_{xb}}$ and the corresponding syntax is as follows

```
REAL var(nxr) [var_1_min, var_1_max;
               var_2_min, var_2_max;
                     ...,       ...;
               var_nxr_min, var nxr_max];
```

for variables of type REAL and

```
BOOL var(nxb);
```

for Boolean variables. Note that according to expression in normal brackets "(" ")", which denotes the dimension of the vector, the lower and upper bounds are specified for each variable in this vector. These bounds are separated by semicolon ";" for each row. If one declares variable in this way HYSDEL inherently assumes a column vector.

Example 9 We want to declare the state real vector $x_r \in \mathbb{R}^2$ where the first variable may vary in $x_{r1} \in [-1,\ 1]$, and the second variable in $x_{r2} \in [0, 1]$. Moreover, one Boolean state is present. The declaration of the STATE section will take the form

```
STATE {
  REAL xr(2) [-1, 1; 0, 1];
  BOOL xb;
}
```

which defines a real vector x_r in dimension 2 and scalar Boolean state x_b.

Note that without specifying dimension of the variable, it is always considered as scalar value.

OUTPUT section

In the OUTPUT section, output variables of the SYSTEM are to be declared. These variables can be of type REAL (i.e. $y_r \in \mathbb{R}$), and BOOL (i.e. $y_b \in \{0,\ 1\}$), similarly as in the INPUT and STATE section. In general, the syntax of the OUTPUT section is as follows

```
OUTPUT { output_item }
```

where each new *output_item* is separated by at least one character space and differs for real and binary variables. The *output_item* for real variables takes the form of

```
REAL var;
```

where the string `REAL`, which denotes the type, is followed by `var` referring to name of the input variable. More variables are delimited by commas "," as in the INPUT and STATE section.

Note that in the OUTPUT section no bounds are specified. It is because the output variable is always considered as an affine function of states and inputs, thus their bounds are automatically inferred.

Example 10 Declaration of the scalar output variables called "y1" and "y2" will be as follows

```
REAL y1, y2;
```

where no bounds are specified.

The *output_item* for Boolean variables takes the form of

```
BOOL var;
```

where the string `BOOL`, which denotes the type, is followed by `var` referring to name of the Boolean state, and semicolon ";" denotes the end of the *state_item*. If there are more output variables, they are delimited by commas ",".

Example 11 Declaration of the scalar Boolean output variables "d1", "d2", and "d3" will take the form

```
BOOL d1, d2, d3;
```

which is the same as in HYSDEL 2.0.5.

Comparing to previous version, output variables may be defined as vectors. Denoting the dimension of the real output vector as n_{yr} and dimension of Boolean output vector n_{yb} allows one to define vectors in the sense of $y_r \in \mathbb{R}^{n_{yr}}$, $y_b \in \{0,\ 1\}^{n_{yb}}$ and the corresponding syntax is then straightforward

```
REAL var(nyr);
```

for variables of type REAL where `nyr` denotes the dimension and

```
BOOL var(nub);
```

for Boolean variables with `nub` specifying the dimension of the column vector.

Example 12 We want to declare the output real vector $y_r \in \mathbb{R}^2$, the second output real vector $q \in \mathbb{R}^2$ and one binary vector outputs $d \in \{0,\ 1\}^3$

```
OUTPUT {
  REAL yr(2), q(2);
  BOOL d(3);
}
```

Note that declaring the binary output in vectorized form is similar to example 11, but in this case is shorter.

PARAMETER section

The PARAMETER section declares variables which will be treated as constants through whole HYSDEL file structure. In this case the syntax of the PARAMETER section is as follows

```
PARAMETER { parameter_item }
```

where each *parameter_item* may take one of the following forms

- declaration of constants

  ```
  type var = value;
  ```

- declaration of constant column vectors with dimension n (the dimension does not have to be present for constants)

  ```
  type var = [value_1; value_2; ..., value_n];
  ```

- declaration of constant matrices with dimensions $n \times m$ (the dimension does not have to be present for constants)

```
type var = [value_11, value_12, ..., value_1m;
            value_21, value_22, ..., value_2m;
                 ...,      ...,  ...,      ...;
            value_n1, value_n2, ..., value_nm];
```

where `type` can be either REAL or BOOL. If there are more parameters with constant values, they have to be separated using semicolons ";" as new variable, i.e.

```
type var1 = value1; type var2 = value2;
type var3 = value3;
```

Notations in vector and matrix description remain the same as in INPUT, STATE and OUTPUT section. That is, each element of the row is delimited with comma "," and the row ends with semicolon ";" .

Example 13 We want to declare constants $a = 1$ as Boolean variable, $b = [-1,\ 0.5,\ -7.3]^T$ and $D = \begin{pmatrix} -1 & 0 & 0.23 \\ 0.12 & -0.78 & 2.1 \end{pmatrix}$ as real variables. This can be done as follows

```
PARAMETER {
  BOOL a = 1; REAL b = [-1; 0.5; -7.3];
  REAL D = [-1, 0, 0.23; 0.12, -0.78, 2.1];
}
```

Moreover, PARAMETER section also allows symbolic parameters of type REAL, assuming that their values will be specified later. If the parameter is symbolic, its declaration reads

```
PARAMETER {
 type var;       /* symbolic scalar */
 type var(n);    /* symbolic vector */
 type var(n,m);  /* symbolic matrix */
}
```

where the string `type` is either REAL or BOOL and the variable `var` can be scalar, vector with n rows, or matrix with dimensions n and m.

Note that dimension of symbolic vectors/matrices, i.e. n, m must not be symbolic parameters.

However, the presence of symbolic variables leads to bad conditioning of the resulting MLD model and therefore it is always required to assign particular values to symbolical expressions before compilation.

If there is a strong need to keep symbolic parameter is MLD models, it is recommended to specify lower and upper bounds on each declared symbolic parameter. The syntax in this case is similar to INPUT, STATE and OUTPUT section,

```
REAL var [var_min, var_max];
```

whereas the declared variable `var` can be only scalar. If there are more symbolic values, they are separated by commas ",".

Note that huge number of symbolic parameters may prolong the compilation time as the number of involved operations is sensitive on symbolic expressions. Furthermore, symbolic expressions have to be replaced with exact values, if the HYSDEL model is going to be further processed.

HYSDEL language supports several numerical expressions, here is example of allowed formats

```
REAL a = 1.101;
REAL tol = 1e-3;
REAL eps = 0.5E-4;
REAL Na = 6.0221415e+23;
```

where the decimal number is separated with dot "." and the decadic power is corresponds to sign "e" or "E", i.e. 2.03×10^{-2} is written as `2.03e-10`. Additionally, HYSDEL language has a set of predeclared variables, to which it suffices to refer with a given string. Precisely,

```
pi; /* pi = 3.141592653 */
MLD_epsilon; /* 1e-6 */
```

and their values can be overridden by the user.

Example 14 We want to declare boolean constant vector $h = [1,\ 0,\ 1]^T$, real matrix $A = \begin{pmatrix} \pi & -0.05 \times 10^2 \\ -0.8 & 12 \times 10^{-3} \end{pmatrix}$, symbolic vector $v \in \mathbb{R}^2$, and symbolic value p which may vary between $[-0.5,\ 1.8]$. The HYSDEL syntax will take the form

```
PARAMETER {
  BOOL h = [1; 0; 1]; /* constant vector */
  REAL A = [pi, -0.05e2; -0.8, 12E-3]; /* constant matrix */
  REAL v(2); /* symbolic vector, will be replaced with real
                value before compilation */
  REAL p [-0.5, 1.8]; /* symbolic parameter, which remains
                         symbolic also after compilation * /
}
```

MODULE section

The MODULE section is a new feature of HYSDEL 3.0.alpha which allows to create subsystems. This is important especially when creating larger HYSDEL structures. To be able to recognize which subsystem is a part of which system a concept of *master* and *slave* files is adopted. A *slave* file will be referred to as a system, which is a part of bigger system, has its own inputs, outputs and acts independently. A *master* file consist of at least one slave file while inputs and outputs are created by subsystems.

Example 15 Example of a standard HYSDEL slave file, without any subsystems (i.e. without the MODULE section)

```
SYSTEM valveA {
/* example of slave HYSDEL file referring to a valveA */
   INTERFACE {
      /* declaration of variables */
      INPUT {  ...  }
      STATE {  ...  }
```

```
      OUTPUT {  ...  }
      PARAMETER {  ...  }
   }
   IMPLEMENTATION {
      /* relations between declared variables */
   }
}
```

The syntax of the MODULE section is given by

```
MODULE { module_item }
```

where each of the *module_item* is composed of

```
name par;
```

The string `name` refers to a name of the subsystem which contains parameter `par`. If there are more parameters, they are separated by commas, i.e.

```
name par1, par2, par3;
```

The syntax is similar to defining symbolical parameters in the PARAMETER section while the exact values have to be assigned in the PARAMETER section before compilation.

Example 16 Assume that the SYSTEM "valveA" creates together with SYSTEM "valveB" a system called tank. The corresponding syntax of the master file in HYSDEL language will look as follows

```
SYSTEM tank {
/* example of master HYSDEL file referring to a tank comprised of two valves */
   INTERFACE {
      /* declaration of variables */
      MODULE {
        valves valveA, valveB;
      }
      INPUT {  ...  }
      STATE {  ...  }
      OUTPUT {  ...  }
      PARAMETER {  ...  }

   }
   IMPLEMENTATION {
      /* relations between declared variables */
   }
}
```

where the string `valves` is the name of the subsystem with parameters `valveA` and `valveB`.

The syntax of the master file `tank.hys` indicates that both of the slave files called `valveA`, `valveB` are in the same directory and created separately. Overall behavior of the system is now described by this master file and the directory listing is shown in Fig. 2.1. If the file `tank.hys` is a part of another system, say it belongs to a storage unit, then this file becomes a slave of the master file, called e.g. `storage_unit` etc. Obviously, such nesting can continue up the desired level.

For instance if one wants to model a production system which consists of several subsystems the syntax will look as follows

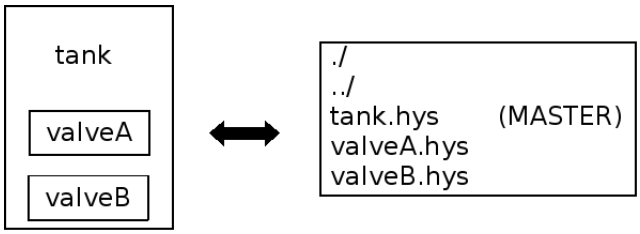

Figure 2.1: Example of a SYSTEM tank containing two subsystems valveA, valveB and corresponding directory listing.

```
SYSTEM production {
 INTERFACE {
   MODULE {
      storage_tank tank1, tank2;
      conveyor_belt belt;
      packaging packer;
   }
   /* other section are omitted */
  }
```

where the slave files are determined via name of the subsystem and corresponding files.

2.2.4 IMPLEMENTATION Section

Relations between variables are determined in the IMPLEMENTATION section. According to type of variables, this section is further partitioned into subsections, which remain the same as in previous version. Syntactical structure of the IMPLEMENTATION section has the following form

```
IMPLEMENTATION { implementation_item }
```

which comprises of curly brackets and *implementation_item*. The *implementation_item* may take only following forms

```
AUX { /* aux_item */ }
CONTINUOUS { /* continuous_item */ }
AUTOMATA { /* automata_item */ }
LINEAR { /* linear_item  }
LOGIC { /* logic_item  }
AD {  /* ad_item */ }
DA {  /* da_item */ }
MUST { /* must_item */ }
OUTPUT { /* output_item */ }
```

where each item appears only once in this section. Each *implementation_item* is separated at least with one space character and may be omitted, if it is not required. The order of each item can be arbitrary, it does not play a role for further processing.

Note that OUTPUT section is also present as in the INTERFACE part but here it has different syntax and semantics.

New changes in the IMPLEMETATION section affect the syntax of each subsection and the common features are listed as follows:

- indexing
- FOR and nested FOR loops
- operators and built-in functions

These changes will be explained first, before describing the individual changes in each INTERFACE subsection.

Example 17 A structure of the standard HYSDEL file is given next where the meaning of each subsection of the IMPLEMENTATION part is briefly explained

```
SYSTEM name {
    INTERFACE {
       /* declaration of variables, subsystems */
    }
    IMPLEMENTATION {
       /* relations between declared variables */
      AUX {
           /* declaration of auxiliary variables, needed for
              calculations in the IMPLEMENTATION section */
      }
      CONTINUOUS {
           /* state update equation for variables of type REAL */
      }
      AUTOMATA {
           /* state update equation for variables of type BOOL */
      }
      LINEAR {
           /* linear relations between variables of type REAL */
      }
      LOGIC {
           /* logical relations between variables of type BOOL */
      }
      AD {
           /* analog-digital block, specifying relations between
              variables of type REAL to BOOL */
      }
      DA {
           /* digital-analog block, specifying relations between
              variables of type BOOL to REAL */
      }
      MUST {
           /* specification of input/state/output constraints */
      }
      OUTPUT {
           /* selection of output variables which can be of type
              REAL or BOOL) */
      }
    }
}
```

Indexing

Introducing vectors and matrices induced the extension of the HYSDEL language to use indexed access to internal variables. The syntax is different for vectors and matrices since it depends on the dimension of the variable. Access to vectorized variables has the following syntax

```
new_var = var(ind);
```

where `new_var` denotes the name of the auxiliary variable (must be defined in AUX section), `var` is the name of the internal variable and `ind` is a vector of indices, referring to position of given elements from a vector. Indexing is based on a Matlab syntax, where the argument `ind` must contain only $\mathbb{N}_+ = \{1,\ 2,\ \ldots\}$ valued elements and its dimension is less or equal to dimension of the variable `var`. Syntax of the `ind` vector can be one of the following:

- increasing/decreasing sequence

  ```
  ind_start:increment:ind_end
  ```

  where `ind_start` denotes the starting position of indexed element, `increment` is the value of which the starting value increases/decreases, and `ind_end` indicates the end position of indexed element.

- increasing by one sequence

  ```
  ind_start:ind_end
  ```

  where the value `increment` is now omitted and HYSDEL automatically treats the value as $+1$

- particular positions

  ```
  [pos_1, pos_2, ..., pos_n]
  ```

  where `pos_1`, `...`, `pos_n` indicates the particular position of elements

- nested indices

  ```
  ind(sub_ind)
  ```

  where the vector `ind` is sub-indexed via the aforementioned ways by vector `sub_ind` with $\mathbb{N}_+$ values

Example 18 In the parameter section were defined two variables. The first variable is a constant vector $h = [-0.5,\ 3,\ 1,\ \pi,\ 0]^T$ and the second variable is a symbolical expression $g \in \mathbb{R}^3$, $g_1 \in [-1,\ 1]$, $g_2 \in [-2,\ 2]$, $g_3 \in [-3,\ 3]$. We want to assign new variables z and v for particular elements of these vectors. Examples are:

- increasing sequence, e.g. $z = [-0.5,\ 1,\ 0]^T$

  ```
  z = h(1:2:5);
  ```

- decreasing sequence, e.g. $v = [g_3,\ g_2]^T$

  ```
  v = g(3:-1:2);
  ```

- increasing by one, e.g. $z = [1,\ \pi,\ 0]^T$

```
z = h(3:5);
```

- particular positions, e.g. $v = [g_1,\ g_3]^T$

```
v = g([1,3]);
```

- nested indexing, e.g. $z = [3,\ \pi]^T$, $k = [2,\ 3,\ 4]$

```
z = h(k([1, 3]));
```

  where the variable `k` has to be declared first.

Indexing of matrices is similar, however, in this case two indices are required. The indexed syntax takes the following form

```
new_var = var(ind_row,ind_col);
```

where `new_var` denotes the name of the auxiliary variable (must be defined in AUX section), `var` is the name of the internal variable, `ind_row` is a vector of indices referring to rows, and `ind_col` is a vector of indices referring to columns.

Note that indexing is based on a Matlab syntax, where the argument `ind` must contain only $\mathbb{N}_+ = \{1,\ 2,\ \ldots\}$ valued elements and its dimension is less or equal to dimension of the variable `var`.

Syntax of the items `ind_row`, `ind_col` is the same as for vectors the item `ind`.

Example 19 In the parameter section a constant matrix is defined

$$A = \begin{pmatrix} 0 & -5 & -0.8 & 1 \\ -2 & 0.3 & 0.6 & -1.2 \\ 0.5 & 0.1 & -3.2 & -1 \end{pmatrix}$$

We may extract values from matrix A to form new variable B as follows

- increasing sequence, e.g.

$$B = \begin{pmatrix} 0 & -5 \\ -2 & 0.3 \end{pmatrix}$$

```
B = A(1:2,1:2);
```

- decreasing sequence, e.g.

$$B = \begin{pmatrix} 0.5 & 0.1 & -3.2 & -1 \\ -2 & 0.3 & 0.6 & -1.2 \\ 0 & -5 & -0.8 & 1 \end{pmatrix}$$

```
B = A(3:-1:1,1:4);
```

- particular positions, e.g. $B = [0,\ 0.3,\ -3.2]^T$

```
B = A(1:3,[1, 2, 3]);
```

- nested indexing, e.g. $B = [0.6,\ -1.2]^T$, $i_{row} = [2,\ 3]$, $i_{col} = [1,\ 2,\ 3,\ 4]$

```
B = h(i_row,i_col(3:4));
```

  where the variables `i_row` and `i_col` have to be declared first.

FOR loops

FOR loops are another important feature of HYSDEL 3.0.alpha version. To create a repeated expression, one has to first define an iteration counter in the AUX section according to syntax

```
AUX {
  INDEX iter;
}
```

where the prefix INDEX denotes the class, and `iter` is the name of the iteration variable. If there are more iteration variables required, the additional variables are separated by commas ",", i.e.

```
AUX {
  INDEX iter1, iter2, iter3;
}
```

As the iteration variable is declared, the FOR syntax takes the form of

```
FOR ( iter = ind ) { repeated_expr }
```

where the string FOR is followed by expression in normal brackets "(", ")" and expression in curly brackets "{", "}". The expression in normal brackets is characterized by assignment `iter = ind` where the iteration variable `iter` incrementally follows the set defined by variable `ind` and this variable takes one of the form shown in section indexing 2.2.4. The expression in curly brackets named *repeated_expr* is recursively evaluated for each value of iterator `iter` and can take the form of

```
aux_item
continuous_item
automata_item
linear_item
logic_item
ad_item
da_item
must_item
output_item
```

depending in which section the FOR loop lies. This allows to use the FOR loop within the whole IMPLEMENTATION section.

Example 20 Suppose, that it is required to repeat *ad_item* in the AD section for each binary variable d_i if state $x_{ri} \geq 0, i = 1,\ 2,\ 3$, i.e.

$$d_1 = x_{r1} \geq 0$$
$$d_2 = x_{r2} \geq 0$$
$$d_3 = x_{r3} \geq 0$$

The iteration index, as well as auxiliary Boolean variable d has to defined first,

```
AUX {
  INDEX i;
  BOOL d(3);
}
```

and they can be consequently used in the AD section as follows

```
AD {
  FOR  (i=1:3) {  d(i) = xr(i) >= 0; }
}
```

HYSDEL 3.0.alpha supports also nested loops. In this case the syntax remains the same, but the *repeated_expr* have now the structure of

```
FOR ( iter = ind ) { repeated_expr }
```

which does not differ from the syntax outlined above.

Example 21 Suppose that we want to code a matrix multiplication for state update equation of the form

$$\begin{pmatrix} x_{r1}(k+1) \\ x_{r2}(k+1) \end{pmatrix} = \begin{pmatrix} 1 & 0.5 \\ 0.2 & 0.9 \end{pmatrix} \begin{pmatrix} x_{r1}(k) \\ x_{r2}(k) \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \end{pmatrix} u_r(k)$$

Assuming that vectors x, u are already declared, we also declare constant matrices in the INTERFACE section

```
PARAMETER {
  REAL A = [1, 0.5; 0.2, 0.9];
  REAL B = [1; 0];
}
```

Secondly, we define iteration counters in IMPLEMENTATION section

```
AUX {
  INDEX i,j;
}
```

and then use the nested loop syntax as follows

```
CONTINUOUS {
  FOR (i=1:2) {
     x(i) = 0;
     FOR (j=1:2) { x(i) = A(i,j)*x(j) + x(i); }
     x(i) = x(i) + B(i)*u;
  }
}
```

Example 22 FOR loops allow also to more complicated expressions. For instance, indexing in a power sequence 2^i where $i = 1, \ldots, 5$. Assume that for given vector $h \in \mathbb{R}^{32}$ it is required to assign a real variable $z \in \mathbb{R}^5$ according to power sequence indexing. In HYSDEL it can be written as follows

```
FOR (i=1:5) { z(i) = h(2^i); }
```

where the variables i, z, and h were previously declared.

Accessing Symbolic Parameters

Symbolic parameters allow more flexibility when creating MLD models. They are declared in the PARAMETER section and in the MODULE section independently, but access is the same. In general, the access is like in Matlab structures. For a standard HYSDEL slave file (which does not have any MODULE section) the syntax takes the common form

```
SYSTEM name {
  INTERFACE {
     /* other parts are omitted */
        ...
     PARAMETER {
       type var;
       type var(n);
       type var(n,m);
       type var [var_min, var_max];
     }
  }
  IMPLEMENTATION {
     /* other parts are omitted */
  }
}
```

and after compilation the symbolic variables are stored in a structure called `S` under fields `params`, i.e.

```
S.params.var
```

If the HYSDEL file contains a MODULE section it is built from subsystems declared in this section. The parameters of the subsystems can be now replaced with concrete values, using the following syntax

```
SYSTEM master {
  INTERFACE {
     /* other parts are omitted */
     MODULE {
     subsystem name1, name2;
     }
     PARAMETER {
       name1.var = value1;
       name2.var = value2;
     }
  }
  IMPLEMENTATION {
     /* other parts are omitted */
  }
}
```

In the remaining IMPLEMENTATION section the symbolical values will be automatically replaced with numerical values.

Example 23 Suppose that a master file `tank` is created by two subsystems `valveA` and `valveB`. Both of the slave file contains symbolical parameters `height` and `width` as in this example for `valveA`

```
SYSTEM valveA {
  INTERFACE {
     /* some parts are omitted */
        ...
     PARAMETER {
```

```
      REAL width, height;
    }
  }
}
```

The exact values can be set in the master file as follows

```
SYSTEM tank {
  INTERFACE {
    /* some parts are omitted */
       ...
    MODULE {
     valves valveA, valveB;
    }
    PARAMETER {
     valveA.width = 0.15;
     valveA.height = 0.23;
     valveB.width = 0.2;
     valveB.height = 0.35;
    }
  }
}
```

HYSDEL operators and built-in functions

Throughout the whole HYSDEL file various relations between variables can be defined. Because the variables may be also vectors (matrices), the list of supported operators is distinguished by

- element-wise operations, in Tab. 2.1
- and vector functions, in Tab. 2.2.

For Boolean variables HYSDEL supports operators, as summarized in Tab. 2.3. Additionally, syntactical functions are available, which allow condition checking and are summarized in Tab. 2.4.

[toto neviem ci ma vobec zmysel udavat, lebo dovolene vyrazy uz mame definovane v *affine_expr*, etc. Ja by som to nasledovne uuplne vyhodil.]

Since the variables may be constants as well as varying, the operators cannot be used arbitrary. This holds especially for varying variables of type REAL (states, inputs, outputs, auxiliary). Denoting this group with notation v, only following relations are allowed

- relations between varying variables v_1 and v_2 of type REAL
  - addition: v_1 + v_2
  - subtraction: v_1 - v_2
- relations between varying variable v and constant parameter p of type REAL
  - addition: $v + p = p + v$ (v and p have the same dimension)
  - subtraction: $v - p = -p + v$ (v and p have the same dimension)
  - multiplication: $v * p = p * v$ (p is scalar)
  - division: $v/p = 1/p * v$ (p is scalar except equal 0)

Relation	Operator	HYSDEL representation
addition	$+$	`+`
subtraction	$-$	`-`
multiplication	$.$	`.*`
division	$/$	`/`
absolute value	$\|a\|$	`abs(a)`
a to the power of b	a^b	`a.^b`
e^b	$\exp b$	`exp(b)`
square root	$\sqrt{a}$	`sqrt(a)`
common logarithm (with base 10)	$\log a$	`log10(a)`
natural logarithm (with base e)	$\ln a$	`log(a)`
binary logarithm (with base 2)	$\log_2 a$	`log2(a)`
cosine	$\cos a$	`cos(a)`
sine	$\sin a$	`sin(a)`
tangent	$\tan a$	`tan(a)`

Table 2.1: List of supported element-wise operators on variables of type REAL.

Relation	Operator	HYSDEL representation
matrix/vector addition	$+$	`+`
matrix/vector subtraction	$-$	`-`
matrix/vector multiplication	$.$	`*`
matrix power	A^b	`A^b`
vector sum	$\sum_i a_i$	`sum(a)`
1-norm of a vector	$\sum_i \|a_i\|$	`norm(a,1)`
∞-norm of a vector	$\max\|a_i\|$	`norm(a,Inf)`

Table 2.2: List of supported vector/matrix operators on variables of type REAL.

Relation	Operator	HYSDEL representation
or	$\vee$	`|` or `||`
and	$\wedge$	`&` or `&&`
one way implication	$\Rightarrow$	`->`
one way implication	$\Leftarrow$	`<-`
equivalence	$\Leftrightarrow$	`<->`
negation	$\neg$	`~` or `!`

Table 2.3: List of supported operators on variables of type BOOL.

Function	HYSDEL representation
true if all elements of a vector are nonzero	`all(a)`
true if any element of a vector is nonzero	`any(a)`

Table 2.4: List of functions for condition checking.

- operations summarized in tables 2.1, 2.2, and 2.4 hold for relations between two parameters p_1, p_2 of type REAL
- for variables of type BOOL are only Boolean expressions valid (Tab. 2.3. However, operations of type REAL can be used but only if the Boolean variable is retyped to class REAL.

]

[Tu treba spomenut evaluaciu prednostnych operacii, ako aj zatvorkovanie - neviem, co presne teraz hysdel podporuje. Na toto sa specialne pytali v ABB].

[If in the expression two or more operators appear, their meaning will be evaluated sequentially, that is, no priorities are between operators. For instance, $v = a + b * c$ the result is $v = (a + b) * c$.]

[If in the expression two or more operators appear, their evaluation is according to operator priority. That is for instance, $v = a + b * c$ will have the result is $v = a + (b * c)$.]

In general, it is recommended to use normal brackets "(", ")" to separate variables into groups and evaluate bracketed and hence prior expressions first. [Example?]

[Operators on symbolical expressions?]

AUX section

In the AUX section one has to declare auxiliary variables needed for derivations of further relations in the IMPLEMENTATION section. The declaration of these variables follows with specifying the type of the variable (REAL, BOOL or INDEX) and defining the name, e.g.

```
type var;
```

where the string `type` can be REAL, BOOL or INDEX and it is followed by a variable `var`. If there are more variables, they are delimited by commas ",".

Example 24 Suppose that in the AUX section we have to declare two iterators i, j, two real vectors $z \in \mathbb{R}^3$, $v \in \mathbb{R}^2$ and Boolean variable $d \in \{0,\ 1\}^5$. The AUX syntax takes the following form

```
AUX {
  INDEX i,j;       /* iteration counters */
  REAL z(3),v(2); /* auxiliary REAL vectors */
  BOOL d(5);       /* auxiliary BOOL vector */
}
```

Contrary to INTERFACE section, here lower and upper bounds on the variables are not given. The values are automatically calculated from variables declared in the INTERFACE section since AUX variables are affine functions of these variables. Moreover, no constants, as well as no parameters are here declared.

CONTINUOUS section

In the CONTINUOUS section the state update equations for variables of type REAL are to be defined. The general syntax is given as

```
CONTINUOUS  { continuous_item }
```

where the *continuous_item* may be inside a FOR loop and takes the form

```
var = affine_expr;
```

The variable `var` corresponds to the state variable declared in the section STATE and it can be scalar or vectorized expression. The *affine_expr* is an affine function of parameters, inputs, states and auxiliary variables, which have been previously declared.

Note that only variables of type REAL, declared in STATE section can be assigned here.

Example 25 Suppose that the state update equation is driven by

$$x(k+1) = Ax(k) + Bu(k) + f$$

where the matrices A, B, f are constant parameters, x is state, u is input. This can be written in HYSDEL language as short vectorized form, i.e.

```
CONTINUOUS {
  x = A*x + B*u + f;
}
```

AUTOMATA section

The AUTOMATA section describes the state transition equations for variables of type BOOL. The general syntax takes the form

```
AUTOMATA  { automata_item }
```

where the *automata_item* may be inside a FOR loop and is built by

```
var = Boolean_expr;
```

The variable `var` corresponds to the Boolean variable defined in the section STATE and it can be scalar or vectorized expression. The *Boolean_expr* is a combination of Boolean inputs, Boolean states, and auxiliary Boolean variables with operators reported in Tab. 2.3.

Note that only variables of type BOOL, declared in STATE section can be assigned here.

Example 26 Suppose that the Boolean state update is driven by following relations

$$x_2(k+1) = u_1(k) \vee (u_2(k) \wedge \neg x_1(k))$$

Related HYSDEL syntax will take the form of

```
AUTOMATA {
 xb(2) = ub(1) | (ub(2) &  ~xb(1));
}
```

LINEAR section

In the LINEAR section it is allowed to define additional variables, which are build by affine expressions of states, inputs, parameters and auxiliary variables of type REAL. In general, the structure of the LINEAR section is given by

```
LINEAR  { linear_item }
```

where the *linear_item* may be inside a FOR loop and takes the form of

```
var = affine_expr;
```

The variable `var` is of type REAL and was previously declared in the AUX section. The *affine_expr* is an affine function of parameters, inputs, states and auxiliary variables of type REAL, which have been previously declared.

Example 27 Consider that it is suitable to define an auxiliary continuous variable $g = -0.5x_1 + 3$ which is an affine function of the state $x \in \mathbb{R}^2$. The variable is firstly declared in the AUX section,

```
AUX {
   REAL g;
}
```

and consequently, the expression in LINEAR section takes the form

```
LINEAR {
   g = -0.5*x(1) + 3;
}
```

Furthermore, the LINEAR section is devoted to determine relations between subsystems and applies only if there is MODULE section present. More precisely, the syntax is given by

```
var = linear_expr;
```

where the variable `var` access the internal input or output variable of type REAL of the declared subsystem. The *linear_expr* is a linear function (i.e. without affine term) of internal inputs or output variables of type REAL of the declared subsystems. [Ma to byt skutocne iba linearna funkcia a nie affinna?] More detailed view for merging of subsystems will be given in special section.

Note that LINEAR section serves also for declaration of interconnection between subsystems (if they are declared in MODULE section).

LOGIC section

LOGIC section allows to define additional relations between variables of type BOOL which might simplify the overall notations. In general the syntax is given by

```
LOGIC  { logic_item }
```

where the *logic_item* may be inside a FOR loop and is built by

```
var = Boolean_expr;
```

The variable `var` corresponds to the Boolean variable defined in the section AUX and it can be scalar or vectorized expression. The *Boolean_expr* is a combination of Boolean inputs, Boolean states, and auxiliary Boolean variables with operators reported in Tab. 2.3.

Example 28 Suppose that we want to introduce the Boolean variable $d = x_1 \wedge (\neg x_2 \mid \neg x_3)$, which is a function of Boolean states $x \in \{0,\ 1\}^3$. We proceed first with variable declaration in AUX section

```
AUX {
   BOOL d;
}
```

and follow with LOGIC section

```
LOGIC {
   d = x(1) & (~x(2) | ~x(3));
}
```

AD section

AD section is used to express the relations between variables of type REAL to Boolean variables only with help of logical operator equivalence. Here, the equivalence operator `<->` is replaced with `=` operator and the syntax is given as

```
AD { ad_item }
```

where the *ad_item* might be inside a FOR loop and it can be one of the following

```
var = affine_expr >= real_num;
var = affine_expr <= real_num;
```

The variable `var` is of type BOOL and has to be declared in the AUX section. The *affine_expr* is a function of states, inputs, parameters and auxiliary variables of type REAL. Operator `>=` denotes the greater or equal inequality ($\geq$) and operator `<=` is less or equal inequality ($\leq$). Expression `real_num` is a real valued number.

Example 29 Suppose that we want to assign to a Boolean variable $d \in \{0,\ 1\}^3$ value 1 if certain inequality is satisfied, otherwise it will be 0.

$$d_1 = \begin{cases} 1 & \text{if} \quad x_1 + 2u_2 \geq 0 \\ 0 & \text{otherwise} \end{cases},\ d_2 = \begin{cases} 1 & \text{if} \quad 0.5x_2 - 3x_3 \leq 0 \\ 0 & \text{otherwise} \end{cases},\ d_3 = \begin{cases} 1 & \text{if} \quad x_1 \geq 1.2 \\ 0 & \text{otherwise} \end{cases}$$

HYSDEL allows to model this behavior using AD syntax

```
AUX {
 BOOL d(3);
}
AD {
  d(1) = x(1) + 2*u(2) >= 0;
  d(2) = 0.5*x(2) - 3*x(3) <= 0;
  d(3) = x(1) >= 1.2;
}
```

Previous version required bounds on auxiliary variables `var` in the form of

```
var = affine_expr >= real_num [min, max, eps];
var = affine_expr <= real_num [min, max, eps];
```

but this syntax is obsolete and related bounds calculation are now obtained automatically. If the bounds will be provided anyway, HYSDEL will raise a warning.

Note that AD section does not use curly brackets "{", "}" to assign the auxiliary variable.

DA section

The DA section defines continuous variables according to if-then-else conditions. HYSDEL 3.0.alpha language supports the following syntax

```
DA { da_item }
```

where the *da_item* might be inside a FOR loop or and it can be one of the following

```
var = { IF cond THEN affine_expr };
var = { IF cond THEN affine_expr ELSE affine_expr};
```

The variable `var` corresponds to auxiliary variable of type REAL, defined in the AUX section. Expression *cond* can be defined as

```
affine_expr >= real_num;
affine_expr <= real_num;
Boolean expr;
```

which denotes certain condition satisfaction. The *affine_expr* is a function of states, inputs, parameters and auxiliary variables of type REAL. Operator `>=` denotes the greater or equal inequality ($\geq$) and operator `<=` is less or equal inequality ($\leq$). Expression `real_num` is a real valued number. The *Boolean_expr* is a function of Boolean states, inputs, parameters and auxiliary variables combined with operators listed in Tab. 2.3.w

Note that if the ELSE string is missing, HYSDEL automatically treats the value equal 0.

Example 30 Suppose that we want to introduce an auxiliary variable z which depends on a continuous state x and a binary input u_b as follows

$$z = \begin{cases} x & \text{if} \quad u_b = 1 \\ -x & \text{if} \quad u_b = 0 \end{cases}$$

HYSDEL models this relation by

```
AUX {
  REAL z;
}
DA {
  z = { IF ub  THEN  x ELSE -x };
}
```

Example 31 It is also possible to define switching conditions using real expressions. Suppose that we want to introduce an auxiliary variable z which depends on continuous states x_1 and x_2

$$z = \begin{cases} 2x_1 & \text{if} \quad x_2 \geq 0 \\ -x_1 + 0.5x_2 & \text{if} \quad x_2 \leq 0 \end{cases}$$

HYSDEL 3.0.alpha allows to models this relation by

```
AUX {
  REAL z;
}
DA {
  z = { IF x(2) >= 0  THEN  2*x(1) ELSE -x(1)+0.5*x(2) };
}
```

and this syntax is more familiar to describe the behavior of PWA systems.

Similarly as in the AD section, previous version required bounds on variables

```
var = { IF cond THEN affine_expr [min, max, eps] };
var = { IF cond THEN affine_expr [min, max, eps]
                ELSE affine_expr [min, max, eps] };
```

This syntax is obsolete and related bounds calculation is now performed automatically. HYSDEL will report a warning if this syntax will be used.

MUST section

MUST section specifies constraints on input, state, and output variables. Regardless of the type of variables, it is required that these condition will be fulfilled for the whole time. The MUST section takes the following syntax

```
MUST { must_item }
```

where the *must_item* can be one of the following

```
real_cond;
Boolean_expr;
real_cond BO Boolean_expr;
Boolean_expr BO real_cond;
```

which denotes certain condition satisfaction. Expression *real_cond* is given by

```
affine_expr >= real_num;
affine_expr <= real_num;
```

and `BO` is a Boolean operator from Tab. 2.3. The *affine_expr* is a function of states, inputs, parameters and auxiliary variables of type REAL. Operator `>=` denotes the greater or equal inequality ($\geq$) and operator `<=` is less or equal inequality ($\leq$). Expression `real_num` is a real valued number. The *Boolean_expr* is a function of Boolean states, inputs, parameters and auxiliary variables combined with operators listed in Tab. 2.3.

Example 32 Consider a system where the one input real variable $u_r \in \mathbb{R}$ and five states $x_r \in \mathbb{R}^5$. Moreover, although the bounds on these variables have been declared, additional constraints are introduced, which are subsets of the declared bounds, i.e. $u_r \in [-2,\ 2]$ and $x_{r1} \in [-1,\ 5]$, $x_{r2} \in [-2,\ 3.4]$, $x_{r3} \in [0.5,\ 2.8]$. These constraints are written in MUST section as follows

```
MUST {
  ur >= -2;
  ur <= 2;
  xr(1:3) >= [-1; -2; 0.5];
  xr(1:3) <= [5; 3.4; 2.8];
}
```

Example 33 For binary variables, one way implications are allowed, as well as equivalence. Example is a system with binary inputs $u_b \in \{0,\ 1\}^2$ which implies that a value of auxiliary binary variable will we 0 or 1, e.g.

$$d_1 \Rightarrow u_{b1}, \quad d_2 \Leftarrow u_{b2}, \quad d_3 \Leftrightarrow x_{b2}$$

and it can be written in HYSDEL as

```
MUST {
  d(1) -> ub(1);
  d(2) <-  ub(2);
  d(3) <-> xb(2);
}
```

Example 34 HYSDEL 3.0.alpha supports also combined REAL–BOOL expressions. Consider a state vector $x \in \mathbb{R}^2$, and binary input $u_b \in \{0,\ 1\}^2$. One may require the constraints as

$$\begin{aligned} u_{b1} &\Rightarrow x_1 \geq 0, \\ x_1 + 2x_2 \geq -2 &\Leftarrow \neg u_{b2}, \\ u_{b1} \vee u_{b2} &\Leftrightarrow x_1 - x_2 \geq 1 \end{aligned}$$

and it can be written in HYSDEL as

```
MUST {
            ub(10)  ->  x(1) >= 0;
     x(1) + 2*x(2) <-  ~ub(2);
     ub(1) | ub(2) <-> x(1) - x(2) >= 1;
}
```

OUTPUT section

The OUTPUT section specifies the output variables for the overall MLD system. Output variables can be both of type REAL and BOOL. The following syntax is accepted

```
OUTPUT { output_item }
```

where the *output_item* might be inside a FOR loop and it can be one of the following

```
var = affine_expr;
var = Boolean_expr;
```

The variable `var` corresponds to a variable declared in INTERFACE part, OUTPUT section which is either of type REAL and BOOL. According to its type, the *affin_expr* is assigned to REAL output and *Boolean_expr* is assigned to Boolean output.

Example 35 Suppose that current system has both real x_r and Boolean states x_b. If we want to write output equations to these states, e.g.

$$\begin{aligned} y_r &= x_r \\ y_b &= x_b \end{aligned}$$

the HYSDEL syntax takes the form of

```
OUTPUT {
  yr = xr;
  yb = xb;
}
```

Note that in OUTPUT section it is not allowed to assign other output variables, as declared in OUTPUT section in INTERFACE part.

2.3 Merging of HYSDEL Files

This section introduces a new feature available in HYSDEL 3.0.alpha which allows merging of hysdel files to subsystems. Initial information about merging was given in the MODULE section but here will be this topic explained more in details.

Merging – INTERFACE part

Similarly as all the variables, subsystems needs to be declared first too. For this purpose a new section of INTERFACE part is created called MODULE. Here are the subsystems declared according their name and containing parameters, as explained in section 2.2.3. Subsystems are independent HYSDEL files with given names, inputs, outpus, states, parameters and are contained in the same directory as the master file.

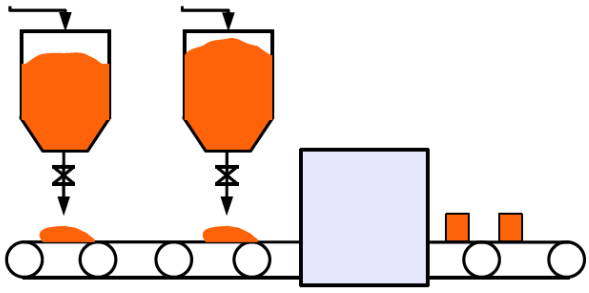

Figure 2.2: Illustrative example of a production system, composed from different parts.

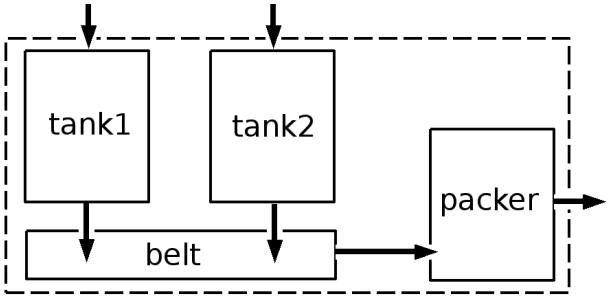

Figure 2.3: The production unit can be represented by four subsystems with corresponding connections.

Example 36 Consider a simple production system which is built by two storage tanks, one conveyor belt and a packaging unit, as illustrated in Fig. 2.2. The task is to model the whole production unit using MLD system. First at all the production unit is virtually separated into subsystems, as shown in Fig. 2.3 and the connections between the subsystems are clarified. Now it is clear which file should be regarded as slave and which as master. We shall call the master file `production` and at first sight it seems that one should declare four slave files, e.g.

1. `tankA`
2. `tankB`
3. `belt`
4. `packer`

but it seems reasonable, to treat the individual parts as symbolic parameters. Subsequently, we can group both tanks into one file and end up in only three subsystems, e.g.

1. `storage_tank`
2. `conveyor_belt`
3. `packaging`

while treating the blocks as symbolic parameters. Declaration of these subsystems in HYSDEL will take a form

```
MODULE {
  storage_tank tank1, tank2;
  conveyor_belt belt;
  packaging packer;
}
```

and it corresponds to three HYSDEL files which will be included, i.e.

1. `storage_tank.hys`
2. `conveyor_belt.hys`
3. `packaging.hys`

Each of these file is a slave file, structured without MODULE section. For instance, the file `storage_tank.hys` may be written in HYSDEL language as follows

```
SYSTEM storage_tank {
 INTERFACE {
   STATE  { REAL level; }
   INPUT  { REAL u; }
   OUTPUT { REAL y; }
   PARAMETER {
      REAL diameter;
      REAL k=1e-2;
   }
 }
 IMPLEMENTATION {
   CONTINUOUS { level = level+1/diameter*(u-k*level);}
   OUTPUT { y = level; }
 }
}
```

Note that in this file some parameters are kept as symbolic and these values have to be set to numerical values before compiling. If the slave files are defined, then we can define the INTERFACE part of the master file as follows

```
INTERFACE {
  MODULE {
    storage_tank tank1, tank2;
    conveyor_belt belt;
    packaging packer;
  }
  STATE { REAL time [0, 1000]; }
  INPUT { REAL raw_flow(2) [0, 5; 0, 5]; }
  OUTPUT { REAL packages; }
  PARAMETER {
    tank1.diameter = 0.3;
    tank2.diameter = 0.5;
    belt.width     = 1;
  }
}
```

where the symbolic parameters are explicitly set. Moreover, the master file has two inputs `raw_flow`, defined as vector, one state `time`, and one output called `packages`.

Merging – IMPLEMENTATION part

In the IMPLEMENTATION part it is required to specify connection between internal subsystems. This can be done in LINEAR section while keeping the standard syntax of *linear_item*. The parameters of the slave files can be accessed like Matlab structures, as mentioned in section 2.2.4. The main change is that inputs will be referred with symbol `u` while outputs with symbol `y` as follows

```
par.u
   .y
```

where `par` denotes the subsystem's name.

Example 37 In this example we will return to example 36 and specify connections according the sketch Fig. 2.3. The IMPLEMENTATION section may be given as

```
IMPLEMENTATION {
 LINEAR {
   tank1.u = raw_flow(1);
   tank2.u = raw_flow(2);
   belt.u  = tank1.y + tank2.y;
   belt.y  = packer.u;
 }
 CONTINUOUS {
   time = time + 1;
 }
 OUTPUT {
   packages = packer.y;
 }
 MUST {
   tank1.level <= 1;
   tank2.level <= 0.8;
 }
}
```

where LINEAR section declares connections between subsystems. The section OUTPUT assigns the variable `packages` to be output from packaging unit which is an overall output variable. Moreover, note that state update equation is also defined, as well as constraints.

2.4 Implementation of new HYSDEL

Implementation of the HYSDEL 3.0.alpha will differ from the standard compilation procedure, as sketched in Fig. 1.1. The core will be replaced with YALMIP package [2] which has many advantages, namely

- direct handling of advanced syntax
- improved condition checking and merging features
- automatic model optimization.

Arising from this tool the compilation will be performed faster that standard HYSDEL compilator.

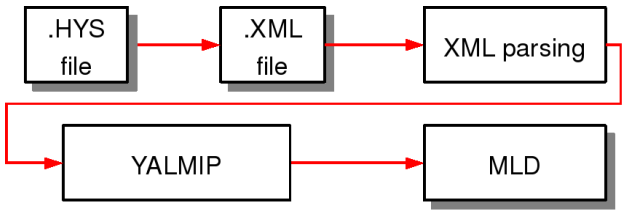

Figure 2.4: New path of generating executable MLD models will rely on YALMIP package.

2.5 EXAMPLES

Example 38 Consider a linear system which contains 2 inputs, 3 states and 1 output with bounded variables

$$\begin{pmatrix} x_{r1}(k+1) \\ x_{r2}(k+1) \\ x_{r3}(k+1) \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 \\ 0.5 & -1 & 0 \\ -0.2 & -0.6 & 0 \end{pmatrix} \begin{pmatrix} x_{r1}(k) \\ x_{r2}(k) \\ x_{r3}(k) \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ -2 & 0.5 \\ 0.3 & -1 \end{pmatrix} \begin{pmatrix} u_{r1}(k) \\ u_{r2}(k) \end{pmatrix}$$

$$y_r(k) = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_{r1}(k) \\ x_{r2}(k) \\ x_{r3}(k) \end{pmatrix}$$

subject to

$$x_r(k) \in \left\{ \begin{matrix} -2 \le x_{r1}(k) \le 2 \\ 1 \le x_{r2}(k) \le 3 \\ 0 \le x_{r3}(k) \le 5 \end{matrix} \right\}, \quad u_r(k) \in \left\{ \begin{matrix} -1 \le u_{r1}(k) \le 1 \\ -1 \le u_{r2}(k) \le 1 \end{matrix} \right\}$$

One may declare the variables in the HYSDEL file in the following way

```
SYSTEM linear_system {
  INTERFACE {
    INPUT {
      REAL ur(2) [-1, 1; -1, 1];
    }
    STATE {
      REAL xr(3) [-2, 2; 1, 3; 0, 5];
    }
    OUTPUT {
      REAL yr(1) [-2, 2];
    }
    PARAMETER {
      REAL A(3,3) = [1, 0, 1; 0.5, -1, 0; -0.2, -0.6, 0];
      REAL B(3,2) = [1, 0; -2, 0.5; 0.3, -1];
      REAL C(1,3) = [1, 0, 0];
    }
  IMPLEMENTATION {
    CONTINUOUS {
      xr = A*xr + B*ur;
    }
    OUTPUT {
      yr = C*xr;
    }
  }
}
```

Bibliography

[1] A. Bemporad and M. Morari. Control of Systems Integrating Logic, Dynamics, and Constraints. *Automatica*, 35(3):407–427, March 1999.

[2] J. Löfberg. Yalmip : A toolbox for modeling and optimization in MATLAB. In *Proceedings of the CACSD Conference*, Taipei, Taiwan, 2004.