ParX Users' Manual

3rd edition

M.G. Middelhoek

December 10, 2017

Contents

Chapter 1

Introduction

ParX is an identification tool for analytical device models. It takes a device model, a set of measurements taken from a real device, and a set of initial values for the unknown model parameters. From these it can determine the optimal values of the parameters and, if requested, an estimate of the model validity domain. These results are obtainted by a proprietary data-fitting algorithm called MODES (from Mode Selection). The main difference between MODES and conventional data-fitting methods is that MODES not only modifies the values of the model parameters, but also the set of measurements. When the algorithm determines that a data point falls outside the validity domain of the model, the point is removed from the data set. Finally, the optimal parameter values are determined using only data points from within the validty domain of the model. Usually, this leads to parameters values that are considderably more consistant with device physics than those obtained by conventional means.

Chapter 2

The ParX command language

The user interface of ParX is text based. When ParX is started, the user is confronted with a '–>' prompt, which indicates that the program is ready to accept commands from the keyboard. All commands are finished by typing a '`;`', after which they are executed by the program. Commands can be extended over several lines by typing a return, after which the user is prompted for additional input with a '+>' prompt. Alternatively, the commands can be read from a file, which can be specified on the command line.

2.1 general structure

ParX commands can be:

- Declarations of ParX variables.
- Operations on ParX variables.
- Inspections or modifications of the ParX program state.

The existence of ParX variables is especially important, since they allow the use of several different data sets, system models and stimulus sets.

Variables come in three types: system, data and stimulus. A *system* variable is used to specify device related information, such as the types, initial values and bounds of the model parameters before extraction. After extraction it holds the extracted values of the parameters. A *data* variable represents a table of measurements for extraction, or holds the results of a simulation. A *stimulus* variable is used to specify the values of the independent interface variables at which a simulation is required.

ParX variables must be declared before they can be used; this is done with one of the declaration commands.

2.1.1 declarations

`sys <var>=<model> ... <var>=<model>;`

Declare variables `var` to be a system with behavior model `model`. The available models differ between installations.

`data <var> ... <var>;`

Declare variables `var` to be data variables.

`stim <var> ... <var>;`

Declare variables `var` to be stimulus variables. Stimulus variables represent ‘measurement’ sets for simulation on the available models. The results of these simulations are stored as `data` variables,

Variables are strings of arbitrary length that match the regular expression `[a-zA-Z][a-zA-Z0-9_.]`. Thus,

```
A
A42
a_silly_name_with_a.dot
```

are all valid variable names.

Apart from this, ParX recognizes the variable ‘`@`’ that represents ‘default’; the actual value depends on the context.

Note that the ParX parser is case sensitive, and that ParX keywords should not be used as variable names.

2.2 variable manipulation

Variables of any type can be copied and deleted.

`<vara> = <varb>;`

Assign the contents of `varb` to `vara`. The variables must be of the same type.

`<var> = @;`

Delete the variable `var`.

2.2.1 system variables

System variables also allow assignments to the parameters fields:

```
<var> = { <pname> unkn = <real> > <real> < <real> };
<var> = { <pname> fact = <real> };
<var> = { <pname> calc = <real> # <real> };
<var> = { <pname> meas = <real> # <real> };
```

Assign a value to parameter `pname` of variable `var`. Variable `var` must be of type `sys`. A parameter can be of one of the following classes:

- `unkn`. The parameter is unknown. For such a parameter a start value, lower bound and upper bound can be specified.
- `fact`. The parameter is exactly known. For such a parameter a value can be specified.
- `calc`. The parameter has been calculated, for example in an extraction. For such a parameter a value and a precision can be specified.
- `meas`. The parameter has been measured. For such a parameter a value and a precision can be specified.

The value, precision, upper bound and lower bound of a parameter need not be specified in the shown order, or even specified at all. Moreover, if the type of a parameter remains the same, it is allowed to omit the keywords `fact`, `unkn`, `calc` and `meas`.

```
<var> = { <pname> @ };
```

Set the given parameter to 'unknown' (`unkn`), and set its start value, lower bound and upper bound to their default values.

2.2.2 stimulus variables

```
<var> = {<tname> = <real> };
<var> = {<tname> > <real> < <real> # <integer> <stepspec> };
```

For stimulus `var`, specify the stimulus of terminal `tname`. The stimulus can be a fixed value or can be stepped between an upper and a lower bound with a given number of steps. The size of each step can be specified with the `stepspec`:

- `lin` The steps are evenly distributed on a logarithmic scale.
- `log` The steps are evenly distributed on a linear scale.

2.2.3 multiple assignment

Assignment commands for a system can become quite complicated expressions, since it must be possible to change only part of the information of a complex data structure. Usually partial assignments are described as complete specifications of a single parameter. It is allowed, however, to give only a partial specification of a parameter. For example:

```
a = { bf = 100 };
a = { bf > 1 };
a = { bf < 10000 };
```

is equivalent to:

```
a = { bf = 100 > 1 < 10000 };
```

Moreover, multiple partial assignments on different parameters can be separated by commas. For example:

```
a = { bf = 100 };
a = { is = 1e-14 };
```

can be replaced by:

```
a = { bf = 100, is = 1e-14 };
```

2.2.4 inspection of variables

```
mod;
sys;
data;
stim;
```

Show a list of the currently defined models, systems, data sets or stimuli.

```
show <var> .. <var>;
<var> .. <var>;
```

Show the current value of the variables `var`. The actual output depends on the type of variable.

2.3 input and output

`<var> < <filename>;`

Read the information in file `filename` into variable `var`.

`<var> > <filename>;`

Write the information in variable `var` to file `filename`.

These commands allow data to be read from files into variables, or written from variables to files. The file names that are used are prefixed with a directory path, and are suffixed with the extensions `.pxd` and `.pxs` for variables of the `data` and `sys` type respectively. Data can also be read from `.csv` and `.json` files, and written to `.json` files.

Declaring a `sys` variable to be of type `model` will attempt to load a model definition file `model.parx`. The default directory path for models is specified in the environment variable `PARX`. Models are by default looked for in de `model` subdirectory. The path must be specified including the final path separator. If the defaults are not wanted, they can be overruled by specifying a complete filename, including path and extension.

2.4 simulation

`sim <stim_var> <sys_var> <data_var>;`

Given a stimulus and a system, simulate the system for all the specified combinations of stimuli. the parameters of the system must all be known (not be a parameter of type `unkn`). The results are stored in the specified data variable.

2.5 extraction

`ext <sys_var> <data_var>;`

Given a system and a data set, assign the extracted parameters to the given system variable. The group information of the data set is also overwritten.

The extraction algorithm that is used is specified by setting the criterion variable.

```
crit = modes
crit = bestfit
crit = chisq
crit = strict
```

Select one of the available fitting algorithms and variants.

- `modes`. Find a set of parameters for the given model and a selected set from the given data set, so that the average distance of all measurements in the selected set is below the specified tolerance.
- `bestfit`. Find a set of parameters for the given model so that the average distance of the measurements in the given data set is minimal.
- `chisq`. Find a set of parameters for the given model and a selected set from the given data set, so that the confidence estimate of the set of parameters for the selected data set is above 0.5. As confidence estimate the χ^2 criterion is used.

  Note that this estimate assumes that the measurement set has a normal distribution in their errors, which is often not true.
- `strict`. Find a set of parameters for the given model and a selected set from the given data set, so that the distance of all measurements in the selected set is below the specified tolerance.

By default the `modes` algorithm is used.

```
tol = <real>;
prec = <real>;
sens = <real>;
tol = @;
prec = @;
sens = @;
```

Set one of the 'tuning' variables of the fit algorithms to the value `real`.

The tuning factors have the following meaning:

- `tol`. Fitting tolerance; The distance between the model curve and the data point ϵ is scaled so that the unit sphere (a sphere with radius 1) represents the precision of the measurements that are used in the fit. The fit is considered successful if the distance of each measurement (`strict`) or the average distance of the measurements (`modes`) ϵ is below 1.

  The tolerance `tol` represents an additional relative precision on all measured values, extending the unit spheres. The default tolerance value is 0.
- `prec`. The precision used in internal calculations. Normally the square-root of the machine precision.
- `sens`. The parameter sensitivity cut-off factor. For each optimization step, the current estimate of the optimum will contain components from all parameters of the model. If, however, the optimum is found to be insensitive for a certain parameter, it contribution is ignored, since it is likely to be inaccurate.

Therefore, the contributions of all parameters that have a sensitivity less than `sens` · `prec` times the sensitivity of the most sensitive parameter are ignored.

By default `sens` is 1, so that the contributions of all parameters that have a sensitivity less than `prec` times the sensitivity of the most sensitive parameter are ignored.

When the special symbol `@` is used, the tuning variable is set to their default value:

`tol`	0.0
`prec`	machine dependent
`sens`	1.0

```
crit;
tol;
prec;
sens;
```

Show the current value of one of the tuning variables.

2.6 global commands

```
read <filename>;
```

Execute ParX commands from the file `filename`.

```
@ < <filename>;
```

Load a ParX state from file `filename` (for debugging).

```
@ > <filename>;
```

Save the ParX state to file `filename` (for debugging).

```
clear;
```

Clear the entire ParX state. Thus, all variables are removed, and the implicit state is set to its default value.

```
trace sim = <int>;
trace ext = <int>;
trace;
```

Set the simulation tracing level (`trace sim`) or extraction tracing level (`trace ext`) to the given value. With the `trace` command without parameters the current tracing levels are shown.

```
exit;
quit;
```

Stop the program.

2.7 command line options

ParX knows the following command line options.

`-e <file>`	Write error messages to file `file`.
`-o <file>`	Write output to file `file`.
`-t <file>`	Write tracing information to file `file`.

Chapter 3

The measurement data structure

This chapter describes the Tm data structures that are expected by ParX for the representation of measurement data.

Data files in the .pxd format are a raw representation of internal Tm data structure. Alternatively, the data can be exchanged using an equivalent JSON format.

Each data file contains one instance of the constructor type datatemplate.

```
datatemplate ::=
    info:string      || general information
    header:[key]     || keys to column captions
    data:[datapoint] || array of data points
;
```

In essence a datatempate represents a table of measurement data, with one data point on each row of the table. These rows are represented by the data field.

Each column in the table is identified by a caption. These captions are listed in the header field. Finally, a general purpose information string is available in the info field.

The captions are represented by instance of the key data type.

```
key == (
    name:string, || caption name
    type:keytype || key type
);
```

The field name represents the name of the data item, and the type the type of data. This is done with the constructor type keytype.

```
keytype ::=
    unkn |    || unknown value
    meas |    || measured value
    calc |    || calculated value
    fact      || factual value
;
```

The header key's are ordered. The sequence should match the columns in the data array.

Data points are represented as a tuple:

```
datapoint == (
    grpid:integer, || group identification number
    crvid:integer, || curve identification number
    rowid:integer, || row identification number
    val:[value],   || array of floating point values
    err:[value]    || array of floating point values
);
```

The `val` array contains the actual values of the various measurements. The `err` array contains the associated measurement errors. The `rowid` contains the identification number of this row. It should be a unique number, but it is not necessary to have sorted or consecutive numbers. The `crvid` can be used to associate a data point with a specific curve, in case of parametric data and for graphical representation of the data. The `grpid` is ignored during input of data. After extraction `grpid` is 1 for data rows that were selected for the fit, and 0 for data rows that were rejected.

Chapter 4

The Model file syntax

ParX models are represented as a set of implicit equations that must be zero in the solution. There is no separation between independent and dependent variables. In combination with the availability of internal (i.e. auxiliary) variables, this allows for a much wider scope of models without the need for internal iteration.

A ParX model file has the `.parx` file extension, and must contain only plain text. Be very careful which editor you use, because editors such as TextEdit are quick to insert special formatting codes, such as smart quotes, which will throw weird syntax errors when the model file is loaded into ParX.

The model description consists of three parts: the header section for the model attributes, the declaration section, and the equations section.

4.1 header section

Each header statement is started by a keyword:

`model:`
: short name of the model, usually the filename with extension

`identifier:`
: expanded description of the model

`author:`
: name of the author and/or copyright notice

`date:`
: modification date of the model

`version:`

version string

The formal syntax rules for the header section are as follows: (Square brackets mark optional terms.)

```
mod[el]:        ”[ <string> ]”
aut[or]:        ”[ <string> ]”
ide[ntifier]: ”[ <string> ]”
dat[e]:         ”[ <string> ]”
ver[sion]:     ”[ <string> ]”
```

4.2 declaration section

The model equations formulate a relationship between several types of symbols:

`variables:`

these are the external variables of the model. They represent the interface with the outside world and can be set and measured.

`auxiliaries:`

these are internal variables of the model, and therefore not open to outside inspection.

`parameters:`

these are the parameters of the model that describe the range of the model behavior. They are available for optimization.

`constants:`

these represent a priori information about the model.

`flags:`

these allow switching between different ?modes? of the model.

`residuals:`

these are the result of the equations and are equal to zero for all solutions of the equations.

`intermediate variables:`

these are temporary variables that can be used to split the model equations in more manageable parts.

All symbols must be declared and defined except for the intermediate variables. They are declared by their first use.

The variables and auxiliaries are defined by three values: their absolute tolerance and two limits. The lower and upper limits are optional, but when specified are enforced in the equations.

The parameters are defined by five values: their default value, default lower bound, default upper bound, lower limit and upper limit. The limits are again optional.

The constants and flags are defined by one value: their default value. Flags are booleans, where zero represents false and non-zero true.

The formal syntax rules:

```
var[iables]:           <symbol> = { [ <abs_tolerance>
                                    [ , <lower_limit>
                                    [ , <upper_limit> ]]] }
                       [ , <symbol> = ... ]

aux[liary | iliaries]: <symbol> = { [ <abs_tolerance>
                                    [ , <lower_limit>
                                    [ , <upper_limit> ]]] }
                       [ , <symbol> = ... ]

par[ameters]:          <symbol> = { <default_value>,
                                    <default_lower_bound>,
                                    <default_upper_bound>
                                    [ ,<lower_limit>
                                    [ , <upper_limit> ]] }
                       [ , <symbol> = ... ]

fla[gs]:              <symbol> = { <default_value> }
                      [ , <symbol> = ... ]

con[stants]:          <symbol> = { <default_value> }
                      [ , <symbol> = ... ]

res[iduals]:          <symbol>
                      [ , <symbol> = ... ]
```

(Note that a limit is identical to a bound with the exception that a limit is checked in code, exceeding a limit will stop the calculating process.)

4.3 equations section

In the equations section the residuals are expressed as a relation between variables. The simulation and extraction algorithms in ParX also require the derivatives of the residuals with regard to the variables and parameters. These derivatives are analytically derived by the model compiler.

Intermediate variables can be used to split the model equations in more manageable parts. These do not require declaration. Using common sub-expressions also leads to a faster implementation by providing better opportunities for optimization of the expressions and their derivatives.

The equation section of the model description is preceded by the keyword `equations:` followed by a number of assignments. These assignments can be to intermediate variables, but must finally be to the residuals. All residuals must be assigned.

The formal syntax rules for the equation section are:

- A line may contain several arithmetic assignments separated by semicolons.
- The last assignment on a line may be terminated by a semicolon but this is not required.
- The available arithmetic operators are: `+`, `-`, `*`, `/`, `^`.
- The available special functions are: `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `exp`, `log` (i.e. `ln`), `log10`, `sqrt`, `abs`, `sign`.
- A special non-arithmetic function is `error(<error_code>)`, which is used for terminating model evaluation in case of out of range conditions.
- Conditional evaluation is also supported:

  ```
  if (<condition>)
      <assignments>
  fi
  ```

  or:

  ```
  if (<condition>)
      <assignments>
  else
      <assignments>
  fi
  ```

  `If (...)`, `else`, `fi` are treated as single special assignments. All keywords must be on a separate line.

- Available logic (conditional) operators are: `==`, `!=`, `<`, `>`, `<=`, `>=`, `&`, `|`, `!`, `not`.

4.4 general syntax rules

- All keywords, function names and constant names are case sensitive and ASCII only.
- All keywords can be shortened to a minimum of three characters.
- Block comments are to be enclosed in '`/*`' and '`*/`'.
- Single line comments are started by '`||`' or '`//`' and run to the end of the line end.
- Spaces and tabs are not significant except in strings.
- Strings must be enclosed in (dumb) double quotation marks '`"`' when spaces are to be preserved.
- Statements are closed by the end of the line. Statements can be extended over multiple lines by placing a continuation mark '`\`' at the end of the line.
- Multiple declarations can be on a single line separated by commas '`,`'.
- Multiple equations can be on a single line separated by semicolons '`;`'.
- All numbers can written using these formats: integer, fixed-point, floating-point, scientific and engineering. Engineering notation uses the following postfix: T = 1e+12, G = 1e+9, M = 1e+6, k = 1e+3, m = 1e-3, u = 1e-6, n = 1e-9, p = 1e-12, f = 1e-15, a = 1e-18
- In declarations the special numbers `inf` or `-inf` are allowed, signifying infinite.

4.5 restrictions

- Maximum line length in the model file: 132
- Maximum number of assignments in equations part of model file: 4096
- Maximum name length (variables, parameters,...): 32
- Nesting of if-statements is allowed with a maximum nesting level of: 16