MDL User Guide and Reference Manual

Author: MDL Development Team

MDL Version 1.0

2017-01-06

Preface

MDL Development Team

Currently Active Developers, and their roles, principle expertise in developing MDL

- Nick Holford (University of Auckland; Uppsala University) : MDL lead, pharmacometric concepts, pharmacology, modelling and simulation tasks, use case development, NONMEM, NM-TRAN to MDL conversion.
- Mike K Smith (Pfizer) : MDL co-lead, documentation, ddmore R package lead, interoperability, model-based drug development, Pharma industry perspective, statistical concepts, use case development, grammar.
- Stuart Moodie (Eight Pillars Ltd / Pfizer) : Language design and implementation, MDL-IDE development, conversion to PharmML
- Maciej Swat (EMBL-EBI) : PharmML lead
- Florent Yvon (EMBL-EBI) : libPharmML lead
- Emmanuelle Comets (INSERM) : Design Object, Optimal Design task specification.
- Paolo Magni (Università degl Studi di Pavia) : Prior Object definition, Bayesian Use Cases, BUGS lead.
- Elisa Borella, Letizia Carrara, Cristiana Larizza, Lorenzo Pasotti, (Università degl Studi di Pavia) : BUGS conversion, BUGS connector, Bayesian Use Cases.
- Zinnia Parra-Guillén (Universidad de Navarra; Freie Universität Berlin) : Training material, documentation, Use Case specification and testing.
- Niklas Hartung (Freie Universität Berlin) : Use Case specification, testing, Monolix.
- Niels Rode-Kristensen (Novo Nordisk) : Interoperability lead
- Richard Kaye (Mango Solutions) : Integration Framework technical lead, SEE builds
- Gareth Smith (Cyprotex) : Common converter
- Henrik Bjugård Nyberg (Mango Solutions / Uppsala University) : PharmML to NM-TRAN translation
- Eric Blaudez (Lixoft) : PharmML to MLXTRAN conversion.
- Nadia Terranova (Merck Serono S.A., a Subsidiary of Merck KGaA) : Standard Output object definition
- Rikard Nordgren (Uppsala University) : NONMEM to Standard Output object translation, PopED conversion.
- Kajsa Harling (Uppsala University) : PsN translation and integration
- Chris Muselle (Mango Solutions) : ddmore R package developer

MDL Contributors

These individuals have made important contributions to MDL and associated deliverables. In alphabetical order:

- *Roberto Bizzotto (CNR Institute of Neurosciences)*
- *Letizia Carrara (Uni di Pavia)*
- *Phylinda Chan (Pfizer)*
- *Marylore Chenel (Servier)*
- *Ronald Gieschke (Roche)*
- *Lutz Harnisch (Pfizer)*
- *Andrew Hooker (Uppsala University)*
- *Mats Karlsson (Uppsala University)*
- *Charlotte Kloft (Freie Universität Berlin)*
- *Natallia Kokash (Leiden Unversity; UCL, London)*
- *Marc Lavielle (INRIA)*
- *Guilia Lestini (INSERM, Paris)*
- *Andrea Mari (CNR Institute of Neurosciences)*
- *France Mentré (INSERM)*
- *Mateusz Rogalski (Mango Solutions)*
- *Maria-Luisa Sardu (Merck Serono S.A., a Subsidiary of Merck KGaA)*
- *Gunnar Yngman (Uppsala University)*

Further Information:

IMI DDMoRe project www.ddmore.eu

DDMoRe Foundation https://www.ddmore.foundation/

MDL http://www.mdl.community/

Report bugs, problems with MDL, ddmore R package, MDL-IDE https://github.com/ModelDefinitionLanguage/website/issues

FAQ about MDL http://www.mdl.community/page/faq/

Download MDL-IDE software: http://downloads.mdl.community/repository/mdl-ide/products/1.6.0/

Download DDMoRe Interoperability Framework Standalone Execution Environment: (SEE) https://sourceforge.net/projects/ddmore/files/install/SEE/

PharmML http://www.pharmml.org/

ProbOnto http://probonto.org

Chapter 1

Introduction

The **M**odel **D**escription **L**anguage (MDL) and the **Pharm**acometrics **M**arkup Language standard (PharmML) (M. Swat et al. 2015) have been developed to convey information about pharmacometrics models and tasks. The goal of each language is to do this consistently between modellers (using MDL) and between software target tools (using PharmML).

MDL is a human writeable and human readable language designed to describe pharmacometric models. It is intended to be largely agnostic about the choice of target tool. MDL should facilitate clear and unambiguous definition of models, with information conveyed in a consistent manner to the PharmML representation and onwards to the target software specific code.

An important concept in the MDL is the separation of data, parameter, model and task descriptions into independent objects rather than combining these in a single file (such as in NONMEM (Bauer 2011)). This supports reuse and interchange of the objects which define each component of the model and related modelling task. This independence means that model objects stored in the DDMoRe Model Repository may be combined with user objects outside the repository e.g. a Model Object, Parameter Object and a {Task Properties Object} may be taken from the Repository and combined with user defined Data Object. This may be useful when a user wishes to assess whether a library model is predictive for their dataset, as a preliminary step before further model refinement.

These facets (target software agnostic code + independence of MDL objects) mean that model definition using MDL is more verbose than code written specifically for any specific target tool. However the principle concept of MDL is that model code is written once and used in many different tools. For estimation, simulation, optimal design. So time spent writing code initially is saved in the longer term since MDL eliminates the need to recode models for different tasks and different software tools.

Why write a new language?

A key deliverable of the DDMoRe project is a unified Model Description Language (MDL), based on established principles, designed to be easily read and written. It is designed to facilitate easy uptake by modellers already experienced in other model definition languages, and will allow the definition of any model-based analysis.

Several languages have been created to support M&S activities. Examples of widely used languages are NONMEM (NMTRAN), Monolix (MLXTRAN) (Lavielle 2012), BUGS (Lunn et al. 2000) and MATLAB (*MATLAB* 2000). However, none are shared, creating difficulties for comparison and integration. Many tools have overlapping functionality, and so the choice of one tool over another is driven largely by user preference, availability of tools, experience of the analyst and whether there is sufficient experience readily available to the analyst to provide support and advice on model building techniques specific for the tool in question. Considerable effort is currently required when moving the model from one software tool to another, as models always have to be recoded in the target software tool language, by hand. A significant need exists to rectify this situation, which DDMoRe is addressing.

Another common situation is using models which were developed by a third party using software that we do not have available. In that case we must try to re-encode the model before we can start using it or developing it further. This can be difficult because we need to ensure that we have all the information to construct the model in a different language. Do we have all the necessary files, settings, subroutines, functions available to us? Are the assumptions used in the model adequately annotated or described in supporting documentation? Do we have understanding of any tool-specific tricks and techniques that allow the model to work in the original software?

MDL provides a user interface to describe models using a common language standard. The aim is that the user writes the model (Model Object) once, in MDL, then uses this model in the tools they require (and have available) in order to complete their M&S tasks, without any tool specific recoding. This interoperability is a core deliverable of the DDMoRe project. (Harnisch et al. 2013)

Additionally, MDL is intended as a standard for communication of models. An analyst who only uses one tool may wish to convey their model to a third party. MDL provides the means to describe the model in a way that is consistent and provides complete information about the model (without any reference to target tool specific code). MDL is designed to focus on describing WHAT the model conveys, rather than focussing on the HOW of implementation. This has an impact on the structure and features used in MDL, but it should aid clarity and reduce ambiguity.

Integrated language standards

As described above, MDL provides the user focused layer of model description. This facilitates user understanding and model sharing between analysts.

PharmML provides the software interchange standard within DDMoRe to facilitate the transfer of models between target tools by ensuring that all of the necessary information about the model is captured and can be translated automatically to any given target tool that has an appropriate PharmML converter.

ProbOnto (M. J. Swat, Grenon, and Wimalaratne 2016) is an ontology and knowledge base that has been developed to describe probability distributions in a consistent and unambiguous way, as well as defining their functions, characteristics and the relationships between distributions.

The Standard Output object (SO) standard provides a consistent format for M & S results and outputs. Its availability as an object within R provides interchange and integration between existing R packages for M & S tasks within the DDMoRe infrastructure.

MDL, PharmML and the SO are the basis for interoperability which is one of the core deliverables of the DDMoRe project.

MDL in use

Very few models can be retrieved from a repository or library, be fit to any given set of data and pronounced valid for inference without further assessment or changes. Thus, the process of fitting models to data, assessing the fit through model diagnostics is an iterative process, culminating in selecting the model which is parsimonious and fit for its purpose in the inferential step decision making, making predictions for future populations of interest, selecting dose or dosage regimen etc. The combination of features in MDL and the ddmore R package facilitates that process. MDL's structure makes changes to data, models, parameters, tasks transparent making it clear exactly which elements are changing and which are constant across steps. Using an R script to define M & S task workflow facilitates an unbroken workflow for a given model and dataset, from exploratory analysis, estimation, diagnostics and simulation across a variety of tools without having to recode the model.

MDL components and structure

The MDL objects are typically defined in a file with extension .mdl. Models may also be stored and retrieved from the DDMoRe Repository either as MDL or PharmML. The key concept in MDL is that these objects can be passed to any target software for use in modelling tasks: estimation, model diagnostics and evaluation, simulation, optimal design.

An overview of the currently specified MDL objects is shown in Figure @ref(fig:MDLOverview).

The MDL is used to specify the inputs and the model used in an M & S task. It does this with four MDL objects defining the model, parameters, data and task properties. An additional object is used to specify the group of objects required for a given task which is known as the Modelling Objects Group (MOG).

The Model Object is the core element of the MDL and Modelling Objects Group (MOG). It defines the mathematical and statistical properties of the model by defining the structural, covariate, variability and observation models. While other objects may change depending on task, the Model Object will typically be unchanged for tasks associated with that model e.g. data visualisation, parameter estimation, model diagnostics, prediction, simulation or optimal design.

The Data Object describes the source of the data and the attributes of each of the data variables. It allows the user to define the inputs to the model and how these inputs and observed data are to be used in definition of the model. It may also be used with data visualisation tools without a Model Object.

The Design Object defines the design parameters interventions, sampling schedules, covariate distributions, populations, study arms - for optimal design or design evaluation, and also for simulation. The Design Object replaces the Data Object in these cases.

The Parameter Object provides values for structural and variability parameters, including bounds on the parameter values for use in estimation. These can be fixed or initial values with associated constraints for parameter estimation or an instantiation of model parameters for use in making predictions or simulations.

The Prior Object defines prior distributions and values of the parameters when performing Bayesian estimation of parameters. It replaces the Parameter Object in this case.

The Task Properties Object contains settings specific to the task which will be passed on to the target software e.g. when estimating parameters it will define the estimation algorithm and associated settings for the algorithm.

It is through combination of the Model Object with other objects that we instantiate the model - linking inputs and observations from the Data Object or Design Object, parameter values from the Parameter Object or Design Object and information about the task settings in the Task Properties Object with the Model Object to form a Modelling Objects Group (MOG) ready for executing a modelling, simulation or optimal design task.

Objects are defined and stored in a MDL file with extension .mdl. It is possible to define more than one Model, Data, Design, Parameter Prior and Task Properties Object within a single MDL file. The MOG Object defines specific individual objects within an MDL file for a given task. Most commonly there will be one object of each type of information in a MDL file used for a task.

Independence of MDL objects

Typically, existing software has used control files that bring the elements in MDL (data, parameters, model and task definitions) together in control, model or project file(s). What is new in the MDL is the concept

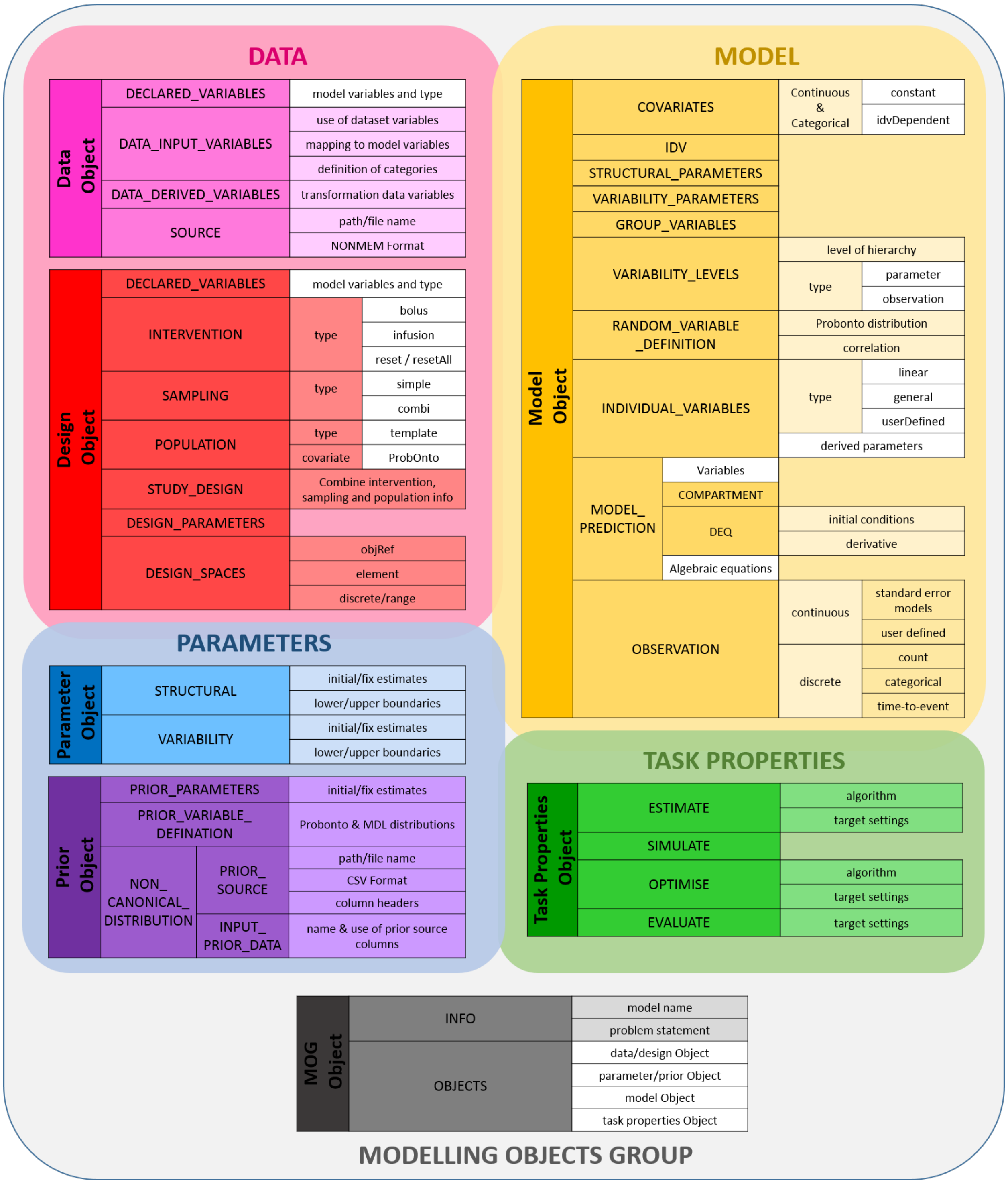

Figure 1.1: MDL Objects

that the elements of the Modelling Object Group (data, parameters, model, task properties) are distinct and independent, allowing the user to combine new data and parameters applicable to their situation with an existing model. Within a M&S task workflow it is easy to see how the core Model Object remains unchanged between estimating parameters, performing model diagnostics, making predictions and simulation future outcomes. The fact that the elements of the MDL are exchangeable also makes it easier to see exactly what elements change between these M & S workflow steps.

Independence of objects also means that the Model Object should be independent of the data and, as a consequence, more easy to read and interpret without needing to have the data to hand. Having an independent Task Properties Object means that the user may store their preferred settings for tasks and target software, suitable for reuse across models and modelling tasks, to facilitate comparison between target software and to ensure reproducibility of results.

Independence of the MDL objects entails defining the contents of each object in isolation. Variables from another MDL object must be declared in the object in which they are referred to e.g. if we need to refer to the Model Object `OBSERVATION` block variable Y in the Data Object `DATA_INPUT_VARIABLES` block then we must declare a matching variable Y in the Data Object.

Task Execution with the ddmore R package

To perform tasks with the model, the user will need to use the ddmore R package. This package contains functions for executing commonly used tasks on the MDL file. The R functions can read and parse MDL objects from a MDL file to create R object representations of MDL, which can then be manipulated within R. These representations of the MDL objects can be combined to form a MOG and then written back to a new MDL file. This means that an MDL file can contain the core Model Object and associated Data, Design, Parameter, Prior and multiple Task Properties Objects which can then be combined into MOGs ready to perform specific tasks. This aids reproducibility since with one MDL file, data set and associated R script the user can perform multiple steps in a pharmacometric workflow for a given model.

Estimation using the estimate function takes as input the user specified MDL file or a MOG Object defined within R. Additional functions allow the user to call modelling tools such as Perl speaks NONMEM (PsN) [8]. Each task produces a Standard Output (SO) object in R which may be the final output or used in subsequent tasks using functions from the ddmore R package or other R packages and commands.

Using R as the language for defining the workflow for M & S tasks with MDL objects allows analysts to tap into existing R packages for performing those tasks. The ddmore package R functions are provided to read and extract information from the SO object and to convert between this, Xpose (Jonsson and Karlsson 1998), mlxR (M. Lavielle, n.d.), PFIM (Bazzoli, Retout, and Mentré 2010) and PopED (Foracchia et al. 2004)R packages. The ddmore R package will extend and enhance what the analyst can do with existing R packages through the common standards that the DDMoRe project brings.

Task properties and settings defined in the Task Properties Object of MDL are distinct from arguments to the R functions for executing tasks. The Task Properties Object provides information to the appropriate target software about the particular settings and options required for a given task. The R function arguments are command line settings or options which are employed when invoking the target software. The Task Properties Object may define the estimation algorithm and associated settings for NONMEM, but the command line options for PsN provided from the ddmore functions govern how NONMEM should be called by PsN. For example, Task Properties specifies an ESTIMATION block with estimation method set to FOCEI, while the arguments of the bootstrap.PsN function in R allow the user to set bootstrap options from PsN such as threads, stratify_on etc. (See PsN bootstrap documentation for more details on PsN bootstrap options).

The MDL Integrated Development Environment

The MDL Integrated Development Environment (MDL-IDE) is a software platform for writing models with MDL. The MDL editor within the MDL-IDE implements the rules of the language through recognising MDL constructs and having a defined grammar and it ensures that MDL models are syntactically correct and result in valid PharmML. The MDL-IDE also provides additional tools giving access to an R editor and console – so that the user can not only develop models, but execute tasks with them and define task workflow through R scripts.

The MDL-IDE gives warnings when the user writes MDL that will result in valid PharmML, but where MDL constructs are used that may not be interoperable. It gives errors when the user writes code that breaks MDL grammar rules and that will result in invalid PharmML.

On interoperability

A key goal of the DDMoRe project is to have an intoperability framework in which models are written in a consistent language, translated to PharmML and from there converted to target software code. Before the DDMoRe project no existing language standard existed across target software used in pharmacometrics modelling, and while the underlying models could be expressed consistently in mathematical and statistical terms, the implementation of any given model varied by tool and by user according to their experience with a given target software tool.

There is ***some*** flexibility within MDL around how the user can express the mathematical and statistical models. Having flexibility allows the user to encode models quickly in a common language (MDL) which can then be shared with others and mutually understood. This flexibility also facilitates encoding in a given target when that language construct does not have a parallel in other tools. **However**, we **STRONGLY** encourage the user to encode the majority of models in a way that will facilitate interoperability. There are MDL constructs that facilitate interoperability these generally appear as built-in functions which translate to specific constructs in PharmML and the target software. These constructs cover many typical models and are designed to allow the user to generate code quickly and have high confidence that it will be interoperable across tools.

The Model Description Language Interactive Development Environment (MDL-IDE) should assist the user in ensuring that the models encoded are valid MDL (and as a consequence, also valid PharmML). Not all models will result in code which can be readily converted to all target tools.

These interoperability constructs will be highlighted in the subsequent sections, but users should pay particular attention to sections on the use of `GROUP_VARIABLES`,`INDIVIDUAL_VARIABLES` and the `MODEL_PREDICTION`.

Evolution of MDL

Development of MDL has been led and influenced by domain experts in M & S, computer language development, system interchange language development (markup languages), and developers of software systems. In developing MDL we have looked at features in established M&S languages, as mentioned above, and aimed to pick out features that will facilitate interoperability, while retaining the flexibility in these languages to describe complex models. The current MDL implementation focusses on interoperability in order to demonstrate that capability. The language standards in MDL, PharmML and SO are the key to eliminating the recoding necessary to pass models between tools used for different M&S tasks.

The MDL language attempts to balance consistency and clarity in definitions, with interoperability and flexibility in translation to PharmML and on to target software. It will continue to evolve to incorporate new features, extending the range of models that can be expressed using the language.

Trying to define a language that maps to all possible models as defined in all possible tools is virtually impossible. However, having a well-defined software interchange standard (PharmML) and mapping MDL into PharmML allows us to focus on describing model features with one target in mind PharmML. The two languages MDL and PharmML - have evolved during the course of the project. The aim is that these two languages should go hand in hand that MDL should convey in an accessible, user (analyst) friendly way, the models that can be encoded in PharmML.

Converter tools then interpret the PharmML rather than the MDL for each software target. Testing this conversion and comparing output downstream allows us to check that the translation results in comparable models.

Future pharmacometrics tools could provide converters to import and export PharmML or use MDL directly as the model specification language. It is our hope that the DDMoRe standards would facilitate more consistency, better understanding of models as well as interoperability between modelling and simulation tools in the future.

Chapter 2

Data Object

The Data Object defines the attributes of the source data file and defines how values in the data source are to be used in the context of a given Model Object. The Data Object is independent of other blocks, including the Model Object. It must ultimately provide appropriate information *to* the Model Object, but as with other Objects in MDL it must be self-contained, so variables from other Objects which are referenced must be declared in the `DECLARED_VARIABLES` block.

The following blocks are defined within the Data Object: `DECLARED_VARIABLES`, `DATA_INPUT_VARIABLES`, `SOURCE`, `DATA_DERIVED_VARIABLES`, `FUNCTIONS`.

Subsetting data for analysis

For quality assurance and audit purposes it is imperative that there be a clear and traceable path between the original data source and data used in analysis. This is normally achieved in one of two ways: through having data manipulation steps performed in a scriptable language to create the dataset for use in analysis, or through having the original data as the input to the analysis and using filtering and subsetting commands in the target analysis software code to define what records are used in the task.

The "NONMEM" method for dealing with outliers and filtering data – commenting out data rows using a specific character as the first item on a data line or specifying conditions under which data is accepted or ignored – is not supported within MDL.

We suggest that users use scriptable languages like R to subset, filter and manipulate data prior to analysis, write the data for analysis to file and then modify the MDL Data Object to reference the appropriate source file in the `SOURCE` block. This aids reproducibility since the data used in estimation will be equivalent across target software tools. If the "ignored" or "dropped" data is saved separately in a file or listing then it will also be possible to quickly and easily verify that the data used in analysis, when combined with the dropped data matches the original dataset.

For legacy data and NMTRAN models, we recommend using the "ignored" function within the "metrumrg" R package (https://r-forge.r-project.org/projects/metrumrg/). This function reads a NONMEM control stream and creates a TRUE / FALSE logical flag for whether the records meet IGNORE and/or ACCEPT criteria specified in the NMTRAN code. This will allow the user to identify which data records have been dropped by NONMEM. Filtering on the original data based on this criteria will allow them to create a dataset ready for analysis with MDL.

For example, if the NMTRAN control file had the following $DATA statement:

```
$DATA mx2007.csv IGNORE=@
  IGNORE ID.EQ.1
  ACCEPT VISI.EQ.3
```

This code means that any data rows which start with “@” are to be omitted, that the subject with ID == 1 should be omitted from analysis, and only VISI == 3 is to be included.

Using the metrumrg function ignored, we have the following code which can be used within the R script. This assumes that the $DATA statements above are in a control file called “run1.ctl”:

```
library(metrumrg)
mx2007 <- read.csv("mx2007.csv", header=T)
mx.dropped <- mx2007[ignored(ctlfile="run1.ctl"),]
mx.kept <- mx2007[!ignored(ctlfile="run1.ctl"),]
```

This separates out the dropped records into the mx.dropped data frame, and the kept records into the data frame mx.kept. The user can then write mx.kept to a .csv file and use this as the file in the `SOURCE` block.

DATA_INPUT_VARIABLES

This block defines the columns in the dataset described in `SOURCE` block and how these map to model variables.

- All columns of the data file defined in the `SOURCE` block must be defined in `DATA_INPUT_VARIABLES` . This aids clarity and readability.
- **In the current MDL , variables in the `DATA_INPUT_VARIABLES` block must be defined *in the column order they appear in the* `SOURCE` block data file.**
- **In the current MDL, variable names in the `DATA_INPUT_VARIABLES` block must *match the names in the header row of the data file* named in the `SOURCE` block. For interoperability with Monolix the case of the names in the header row must match the case of the data variable in the `DATA_INPUT_VARIABLES` block.**
- `DATA_INPUT_VARIABLES` cannot have more than one `use` defined.
- All `DATA_INPUT_VARIABLES` must have a `use` defined.
- Columns in the data which are not required in the model should have `use is ignore`.

The typical syntax for defining items in the `DATA_INPUT_VARIABLES` block is:

```
<Variable name> : { use is <use type> }
```

Define types for `use type` are:

Use	**Defines**
id	Individual identifier. Typically subject ID in clinical trials. Defines the indvidual level of parameter variability.
idv	Independent variable. Typically TIME. In the Model Object, the reserved variable T is used as the integrator va
amt	Dose amount
dv	Dependent variable
varLevel	Defines a level of variability in the model. Not required for `DATA_INPUT_VARIABLES` with `use is id` or `use is`
covariate	Covariate for use in definition of fixed effect variable
catCov	Categorical covariate for use in definition of fixed effect variables based on categories
variable	Defines any model input not covered by other types. Corresponds to the “regressor” variable type in Monolix.
dvid	Dependent variable identifier when there is more than one type of observation.
evid	Event identifier. This type is defined in MDL, but not yet implemented by converters in the current SEE.
mdv	Missing dependent variable
cmt	Compartment identifier
ss	Steady State indicator
ii	Inter-dose interval i.e. time between repeated doses

Use	**Defines**
addl	Number of additional doses in repeated dose administration
rate	Zero-order input rate
ignore	Ignores a data variable

The `use` types correspond to those used by NONMEM and Monolix. `varLevel` corresponds to OCC in Monolix, `catCov` corresponds to CAT in Monolix. (NMTRAN does not have equivalent reserved words for these types of variables – their use is implicit in the implementation of the model).

Use `use is covariate` and `use is catCov` for data variables that are to be used in "linear" definition of `INDIVIDUAL_VARIABLES`. Note that covariates and categorical covaraites used in "linear" definitions within the `INDIVIDUAL_VARIABLES` block should not vary with time, although they may vary by occasion.

Use `use is variable` for data variables that are to be used directly in the calculations within the `MODEL_PREDICTION` block – for example, individualised PK parameter predictions for use in a PD model. These may be time-varying covariates or inputs to the `MODEL_PREDICTION` equations.

`use is variable` corresponds to the "regressor" variable type in Monolix.

MDL does not have "reserved" names for variables in the Model Object other than the default for the independent variable T. The intended use for variables is defined via the `use` is . . . " attribute as described above. The choice of names for `DATA_INPUT_VARIABLES` should be meaningful to the user and clear for any third party reading the code.

Defining dose amount

The dosing amount is defined through `DATA_INPUT_VARIABLES` with `use is amt`. It is assumed that when the column has a non-zero / non-missing value, this amount is assigned to the relevant Model Object dosing variable.

The Model Object variables to which dose is assigned should be declared in the `DECLARED_VARIABLES` block and should have type::dosingTarget. This applies equally when doses are assigned to differential equation variables (UseCase1), PK input compartments with `type is depot` or `type is direct` (UseCase4) or variables in analytical models (UseCase2).

The syntax for defining the dose amount is:

```
<Variable> : { use is amt, variable = <mdlObject variable>}
```

For example:

```
  AMT : { use is amt, variable = GUT }
```

Within the Model Object, the dose amount is defined when AMT > 0 if the `DATA_INPUT_VARIABLES` variable is defined as `use is amt`. It is not necessary to conditionally assign a Model Object variable to this value when AMT > 0. This is taken care of in translation to PharmML behind the scenes. If, however, the user chooses to treat the dose amount column as `use is covariate` or `use is variable` then this **will** need conditional assignment within the model. Also, in that case the dosing amount variable should be declared in the `COVARIATES` block of the Model Object.

For analytical models (such as UseCase2) the dosing amount may be defined with respect to a dosing variable in the Model Object, rather than as an initial amount in a differential equation or compartment. In that case it is necessary to declare the dosing variable within the `MODEL_PREDICTION` block:

The declared dosing variable for analytical models should have type ::dosingTarget. This maps into variables of type ::dosingVar and ::dosingTarget in the Model Object. Examples of ::dosingTarget are `COMPARTMENT` variables with "type is depot", "type is direct" and also variables defined by differential equations.

For example:

In the Data Object:

```
DECLARED_VARIABLES{ D::dosingTarget }

DATA_INPUT_VARIABLES{
  AMT : { use is amt, variable=D }
  }
```

In the Model Object:

```
MODEL_PREDICTION {
  D :: dosingVar # dosing variable
  k = CL/V
  CC = if ( T < TLAG) then 0
    else (D/V) * KA/(KA-k) * (exp(-k * (T - TLAG))- exp(-KA*(T-TLAG)))
 }
```

MDL supports definition of multiple doses via `DATA_INPUT_VARIABLES` defined as `use is ii`, `use is addl`, `use is ss`.

MDL supports infusion rate (zero-order input rate) specification via `DATA_INPUT_VARIABLES` with `use is rate`. **The current version of MDL does not support negative values for "use is rate" in order to allow model defined rate or duration.**

Please also read the section @ref(assignment-to-a-single-variable-using-variable) and section @ref(assignment-to-multiple-variables-using-define) on assignment using `variable = ...` compared to using `define = ...`.

Defining the independent variable

The independent variable is defined through a `DATA_INPUT_VARIABLES` with `use is idv`. The Model Object has an `IDV` block where the independent variable in the model is defined and the model variable defined in this block is automatically mapped to the Data Object variable defined as `use is idv`. This is used to link the model independent variable in the Model Object `IDV` block and the event (observation, dosing) times specified in the `DATA_INPUT_VARIABLES`.

Syntax:

```
<Variable> : { use is idv }
```

Typically the independent variable will be time. UseCase11 shows a pharmacodynamic model with the plasma concentration as the independent variable.

Defining the dependent variable

The dependent variable is defined through a `DATA_INPUT_VARIABLES` with `use is dv`. This data variable is the observation and must be mapped to the Model Object block prediction variable using the Data Object block.

Continuous data, single Model Object prediction

If there is only one observation type and it is continuous then the user should map the `DATA_INPUT_VARIABLES use is dv` variable to a prediction variable name in the Model Object block. It is possible to map the dependent variable to a single, specified Model Object prediction variable using `variable = <NAME>`

The Model Object block prediction variable name must be declared in the Data Object block.

The observation variable in the `DECLARED_VARIABLES` block must have type ::observation.

For example:

In the Data Object:

```
DECLARED_VARIABLES{ Y::observation }

DATA_INPUT_VARIABLES{
  ...
  DV : { use is dv, variable = Y }
  ...
  }
```

In the Model Object:

```
OBSERVATION{
  Y : {type is additiveError, additive=SD_ADD, eps=EPS_Y, prediction= CONC}
  }
```

Multiple Model Object `OBSERVATION` predictions

If mapping the single dataset dependent variable column (with `use is dv`) to multiple Model Object `OBSERVATION` prediction variables, it is necessary to also define a `DATA_INPUT_VARIABLES` variable with `use is dvid` that identifies which records of the dataset belong to each observation type. The user must also define how to map the Model Object block variables to values in the `DATA_INPUT_VARIABLES`with `use is dvid`.

The syntax is as follows `define = {<value> in <data column name with "use is dvid"> as <`[DECLARED_VARIABLES]` variable>, etc.}`.

For example (UseCase3):

In the Data Object:

```
DECLARED_VARIABLES{ CP_obs::observation PCA_obs::observation }

DATA_INPUT_VARIABLES{
  ...
  DVID : { use is dvid }
  DV : { use is dv, define={1 in DVID as CP_obs, 2 in DVID as PCA_obs} }
  ...
  }
```

In the Model Object:

```
OBSERVATION{
  CP_obs : {type is combinedError1,
    additive = RUV_ADD,
    proportional = RUV_PROP,
    eps = EPS_CP,
    prediction = CC }

  PCA_obs : {type is additiveError,
    additive = RUV_FX,
    eps = EPS_PCA,
    prediction = PCA }
  }
```

This means when the data variable with `use is dvid` has the value 1 then the observation in the data variable with `use is dv` ***within the same data record*** will be mapped to the Model Object block variable CP_obs, and when this variable has the value 2 it will be mapped to PCA_obs.

All continuous observation variables must be declared in the `DECLARED_VARIABLES` block as ::observation.

When declaring an outcome variable that is binary (UseCase12) or categorical (UseCase13), it is necessary in the `DECLARED_VARIABLES` block to define the category name for the Model Object variables which define the prediction.

For example (UseCase12):

In Data Object:

```
DECLARED_VARIABLES{ Y withCategories {none, event} }
```

Using this convention, the type is implicit in the "withCategories" keyword.

Note that when declaring an outcome variable with a Poisson count or Binomial number of successes outcome, Y is declared as the variable attribute of an anonymous OBSERVATION block list.

For example (UseCase11):

In the Data Object:

```
 DECLARED_VARIABLES{ Y::observation }
```

In the Model Object:

```
RANDOM_VARIABLE_DEFINITION(level = DV){
  Y ~ Poisson1(rate = LAMBDA)
  }

OBSERVATION{
  :: {type is count, variable = Y}
  }
```

Mapping data variables to model variability levels

The hierarchy of model levels of variability is defined within the Model Object in the `VARIABILITY_LEVELS` block. Within the Data Object we match the levels in the model to `DATA_INPUT_VARIABLES`to identify the data variables where changing values signify new individuals, occasions or observations (and other levels of variability in the model). By default the lowest level of the hierarchy is the observation level with `DATA_INPUT_VARIABLES` defined as `use is dv`. The other variability level commonly used is the experimental unit, in clinical trials this is typically the subject with `DATA_INPUT_VARIABLES` defined as `use is id`.

The level in the hierarchy is defined by a numerical level value in the [Model Object] [`VARIABILITY LEVELS`] block.

For example (UseCase1):

In Data Object:

```
DATA_INPUT_VARIABLES{
  ID : { use is id }
  ...
  DV : { use is dv, variable = Y }
  ...
  }
```

In the Model Object:

```
VARIABILITY_LEVELS{
  ID : { level =2, type is parameter }
  DV : { level =1, type is observation }
 }
```

Other variability levels may be defined in `DATA_INPUT_VARIABLES` as `use is varLevel`. This will be used to define variability levels such as occasion, study (if modelling across more than one study) etc.

For example, when defining occasions for use in between occasion variability models:

```
  OCC : { use is varLevel }
```

For example (UseCase8):

In Data Object:

```
DATA_INPUT_VARIABLES{
  ID : { use is id }
  TIME : { use is idv }
  WT : { use is covariate }
  AGE : { use is covariate }
  SEX : { use is catCov withCategories {female when 1, male when 0} }
  AMT : { use is amt, variable = INPUT_KA}
  OCC : { use is varLevel }
  DV : { use is dv, variable = Y }
  MDV : { use is mdv }
  }
```

In Model Object:

```
VARIABILITY_LEVELS{
  ID : { level = 3, use is parameter }
  OCC : { level = 2, use is parameter }
  DV : { level = 1, use is observation }
  }
```

MDL does not place any limits on the number of levels of variability within the Model Object. However some constraints may exist within the target software used for estimation.

Note that occasionally, if modelling summary level data in a model-based meta-analysis, the experimental unit defined in the data may be different, for example, the treatment arm rather than an individual subject.

For example:

In Data Object:

```
DATA_INPUT_VARIABLES{
  ARM : { use is id }
  DV : { use is dv, variable = Y }
  }
```

In Model Object:

```
VARIABILITY_LEVELS{
  ARM : { level=2, type is parameter }
  DV : { level=1, type is observation }
  }
```

See the section on `VARIABILITY_LEVELS` for discussion of identifying the reference level of variability when using a model with a Design Object.

Defining covariates

Defining continuous covariates

Continuous covariates (including time-varying covariates) are defined as `use is covariate`.

Note the discussion in the sections on `COVARIATES` and `INDIVIDUAL_VARIABLES` about the requirements of covariates that are to be used in definition of `INDIVIDUAL_VARIABLES`.

Interpolation can be specified for continuous covariates through the argument:

```
interp = &constInterp | &cubicInterp | &lastValueInterp |
&linearInterp | &nearestInterp | &pchipInterp | &splineInterp
```

Note the "&" in front of the interpolation type to signify that a function is being referenced.

Interpolation functions take as arguments t0 (start of time interval), t1 (end of time interval), x (variable for interpolation), x0 (variable value at t0) and x1 (variable value at t1).

Defining categorical covariates

Categorical covariates are defined as `use is catCov` and must have a mapping between the values in the data column and categories to be used in the model. This serves a dual purpose: firstly, providing clarity on how numeric codes in the data map to named categories and secondly, transparency in model description by allowing us to use those named category labels in the model when referring to the categories.

The mapping is performed by using the keyword `withCategories` and then specifying the mapping between categories and values using when in a comma separated list.

```
  SEX : {use is catCov withCategories {female when 1, male when 0} }
```

The categories defined (female, male) must match those defined within the Model Object block. Note that the category labels (female, male) are not character strings. They are enumerated variables, and can be referred to in the model as SEX.female or SEX.male. For example to define an action dependent on the SEX being female in the Model Object we would use the Boolean comparison SEX == SEX.female. This evaluates to true when the value in the SEX column matches the value defined (in the `DATA_INPUT_VARIABLES` as above) that corresponds to the female category of the SEX variable.

In the current version of MDL the categories defined must be unique i.e. it is not possible to assign more than one value to a category, nor is it possible to define several categories with the same name. This means that each category maps to only one data value. The following is code NOT valid:

```
 food : {use is catCov withCategories {fed when 2, fasted when 1, fasted when -999}
```

Nor is:

```
 food : {use is catCov withCategories {fed when 2, fasted when [-999,1] }
```

The user should identify all categories in the DATA_INPUT_VARIABLE block:

```
food : {use is catCov withCategories {fed when 2, fasted when 1,
  missing when -999}
```

Defining model inputs or time-varying covariates

Often we will want to pass data variables to the `GROUP_VARIABLES` or `MODEL_PREDICTION` blocks in the Model Object which will not fit the definition of covariates as defined in section @ref(defining-covariates). This might be the case if we want to pass individualised predictions of PK parameters into a PD model, or a time-varying covariate such as plasma concentration or age for use in a maturation model. These variables are sometimes referred to as regressors or model input variables.

These variables are defined in MDL as `use is variable"`.

Variables with `use is variable` should not be used with definition of `INDIVIDUAL_VARIABLES` with `type is linear`.

Assignment to a single variable using `variable`

If the value of the variable within the data is to be mapped to a single model variable e.g. dosing amount D, then the variable attribute must be assigned, and the associated variable declared in `DECLARED_VARIABLES`:

```
DECLARED_VARIABLES{ D::dosingTarget Y::observation }
  DATA_INPUT_VARIABLES{
  ...
  AMT : { use is amt, variable = D }
  DV : { use is dv, variable = Y }
  ...
  }
```

Assignment to multiple variables using `define`

In the case of mapping data values to ***multiple*** variables depending on the values of another variable, the syntax is as follows `define = {<value> in <data variable name> as <declared_variables>, etc.}`.

The Model Object variables used in this definition must also be declared in the `DECLARED_VARIABLES` block.

```
DECLARED_VARIABLES{ CP_obs::observation PCA_obs::observation }
  DATA_INPUT_VARIABLES{
  ...
  DVID : { use is dvid }
  DV : { use is dv, define={1 in DVID as CP_obs, 2 in DVID as PCA_obs} }
  ...
  }
```

SOURCE

This block defines the source data file for use with the model. It defines the file name and file format.

In the current version of MDL it is assumed that the `SOURCE` data file will be present as an ASCII comma-delimited text file (.csv). We also assume that the dataset conforms to NONMEM data standards. The data file should have a header row with names matching those in the `DATA_INPUT_VARIABLES` block. Data values should be numeric. Missing values should be denoted by ".".

The MDL syntax is as follows:

```
<source object name> : {file = <filename>, inputFormat is nonmemFormat }
```

For example:

```
SOURCE {
  srcfile : {file = "warfarin_conc.csv",
    inputFormat is nonmemFormat }
  }
```

For the current version of the interoperability framework SEE, data files must be in the same folder and workspace as the model file.

`DECLARED_VARIABLES`

The `DECLARED_VARIABLES` block is used in the Data Object and in the Design Object, and it can be used in the Parameter Object when required to define correlations or covariances.

This block links variables defined in the one MDL Object with variables defined within another MDL Object, typically referring to model variables in the Model Object which need to be referenced in another object, like the Data Object, Design Object or Parameter Object.

For example: When defining the Pharmacokinetic model (UseCase1) we need to define which Model Object variable receives the dosing amount – in this case the differential equation specifying the amount in the GUT, and an observation variable Y.

Since the MDL objects are independent of each other (e.g. the Data Object is not "aware" of Model Object variables) we must explicitly declare Model Object variables within other MDL objects if we need to refer to them.

When declaring the variable, the user must also define the variable type. A controlled vocabulary of types is provided through the MDL-IDE. Type is defined via the double colon:

`<Variable name> :: <type>`

For example:

`GUT :: dosingTarget Y :: observation.`

This improves validation of the MOG Object since we can check that Data Object use is mapped to an appropriate type of variable and that this passes through to the appropriate type of variable in the Model Object. This change is also required to ensure that the Data Object and the Design Object handle definition and declaration of variables in an equivalent way.

The following types will be typically of use in `DECLARED_VARIABLES` :

`dosingTarget`, `OBSERVATION`.

As has been described above, when declaring an outcome variable that is binary (UseCase12) or categorical (UseCase13), it is necessary in the `DECLARED_VARIABLES` block to define the categories for the outcome variable.

Note that when declaring an outcome variable with a Poisson count or Binomial number of successes outcome, Y is declared as observation.

Note that no delimiter is required between variables defined in the `DECLARED_VARIABLES` block.

In the current MDL, variable names mapped across MDL Objects must match e.g. if we declare variable Y in the Data Object, then this will be linked to the Model Object block variable Y.

`DATA_DERIVED_VARIABLES`

Occasionally, the user will need to define new variables that depend on existing information in the dataset for example the dose amount or dose time when this is to be used as a covariate in the model. The `DATA_DERIVED_VARIABLES` block allows the user to define new variables using data in the columns with `use is amt` and `use is idv`. The variables defined must have unique names, different from those specified in the `DATA_INPUT_VARIABLES`.

Syntax is

```
<variable> : {use is <doseTime / covariate / variable / doseInterval>, ... }
```

The subsequent arguments of the list depend on its use.

```
<variable> : {use is doseTime,
  idvColumn = <DATA_INPUT_VARIABLE variable with "use is idv">,
  dosingVar = <dosing variable defined with type ::dosingTarget>}
```

In the current MDL, `use is doseInterval` below cannot currently be mapped to PharmML.

```
<variable> : {use is doseInterval, idvColumn =
  <DATA_INPUT_VARIABLE variable with 'use is idv'>,
  dosingVar = <dosing variable defined with type ::dosingTarget> }

  <variable> : {use is covariate, column = <DATA_INPUT_VARIABLE variable> }

  <variable> : {use is catCov, column = <DATA_INPUT_VARIABLE variable> }

  <variable> : {use is variable, column = <DATA_INPUT_VARIABLE variable> }
```

For example (UseCase2_1):

```
DATA_DERIVED_VARIABLES{
 # Like 'use is amt' we assume that the DT variable is only assigned
 when AMT > 0.

 # The typing ensured that the attributes reference a column with the
 correct 'use'.

 DT : { use is doseTime, idvColumn=TIME, amtColumn=AMT }

 }
```

Chapter 3

Parameter Object

The Parameter Object defines model parameter values for use with the Model Object. In estimation tasks, these are typically the initial values for the estimation algorithm, or fixed parameter values within the model.

The Parameter Object should provide a value for each parameter listed in the Model Object `STRUCTURAL_PARAMETERS` and `VARIABILITY_PARAMETERS` blocks.

`STRUCTURAL` and `VARIABILITY` parameter blocks are kept separate to allow the user to quickly identify the function of each parameter in the model and to facilitate certain tasks, for example fixing variability parameters for simulation.

STRUCTURAL

The `STRUCTURAL` block defines the numerical values of the structural parameters with optional constraints (low and high values) and whether the value is fixed or to be estimated. Each structural parameter must have the `value` argument assigned a numeric value.

For each structural parameter the typical construct will be

```
<PARAMETER NAME> : { value = <numeric> }
```

Or, with additional optional attributes

```
<PARAMETER NAME> : { value = <numeric>, lo = <numeric (lower bound)>,
                      hi = <numeric (upper bound)>, fix = <true | false> }
```

This provides a numerical value for a parameter which may be used as an initial estimate for estimation or as a value for simulation. Numerical values may be expressed in scientific notation.

The `lo` and `hi` attributes are optional and are used to define lower and upper boundaries for estimation.

The `fix` attribute is optional. It may be set to a logical value of true or false. The default value of `fix` is false. When `fix` is true the parameter will not be estimated in an estimation task. Specifying `fix = true` overrides any setting of `lo` and `hi`.

```
STRUCTURAL {
 POP_CL : { value = 0.1, lo = 0.001 }
 POP_V : { value = 8, lo = 0.001 }
 POP_KA : { value = 0.362, lo = 0.001 }
 POP_TLAG : { value=1, lo=0.001 }
 BETA_CL_WT : { value = 0.75, fix = true }
 BETA_V_WT : { value = 1, fix = true }
 RUV_PROP : { value = 0.1, lo = 0 }
 RUV_ADD : { value = 0.1, lo = 0 }
  } # end STRUCTURAL
```

Note on parameter values

It is typical to specify log-Normal distributions for parameters, but the user should be aware that in some models, parameters may be negative. As with other languages, the user should be careful to avoid parameterisations that would lead to taking logs of a negative number.

VARIABILITY

The `VARIABILITY` block defines the names and values of random effect parameters (including covariance or correlation parameters) that are to be used in the Model Object. Similar to the `STRUCTURAL` block above, the `VARIABILITY` block provides initial values for estimation. Each variable must have the `value` argument assigned a numeric value.

Similar to the `STRUCTURAL` block, the `VARIABILITY` block requires attributes for each random effect used in the model.

For each random effect parameter the typical construct is

```
<PARAMETER NAME> : { value = <numeric> , type is <sd | var> }
```

With additional `fix` attribute

```
<PARAMETER NAME> : { value = <numeric>, type is <sd | var>, fix = true }
```

The `type` argument specifies whether the initial values ***and*** parameter estimation are specified on the standard deviation scale.

Within the Parameter Object `VARIABILITY` block, we no longer need to specify the type of variability (variance or sd). The Parameter Object simply defines values for the parameters used in the Model Object block. The user then needs to ensure that the parameter values are on the appropriate scale.

Note: that in the version of PsN used in the current version of the SEE, bootstrap estimates of variability parameters are not available on the standard deviation scale. Returned variability parameters from bootstrap estimation will be on the *variance* scale.

An example `VARIABILITY` block:

```
VARIABILITY {
 PPV_CL : { value = 0.1, type is sd }
 PPV_V : { value = 0.1, type is sd }
 CORR_CL_V : { value = 0.01 }
 PPV_KA : { value = 0.1, type is sd }
 PPV_TLAG : { value = 0.1, type is sd, fix=true }
 RUV_PROP : { value = 0.1, lo = 0 }
 RUV_ADD : { value = 0.1, lo = 0.0001 }
 } # end VARIABILITY
```

Note that parameter estimates for residual errors e.g. RUV_PROP and RUV_ADD above are now specified as `VARIABILITY` parameters, and not `STRUCTURAL`. Typically, these parameters are defined as multipliers of standard Normal(0,1) random variables in definition of the residual error model. The purpose of the `VARIABILITY` block definition in the Parameter Object is to make it easier to identify those parameters associated with variability and if required turn off variability by fixing these parameters to zero. Placing the residual error parameters in the `STRUCTURAL` parameter block made this difficult, since these parameters can have arbitrary names. Nevertheless, the models are equivalent whether residual errors are defined by parameters specified in the `STRUCTURAL` or in the `VARIABILITY` block.

Note also that correlations between parameters are given parameter values in the `VARIABILITY` block, but definition of correlation and covariance now occurs in the Model Object `RANDOM_VARIABLE_DEFINITION` block.

When defining values for multivariate distributions, the user may need to define vectors and matrices to define the mean and covariance matrix (respectively). To do so the user defines a list with the following syntax:

To define a vector of length k:

```
<VARIABLE NAME> : {vectorValue = [ <value1>, <value2>,  <valuek>] }
```

The square brackets denote that the result is a vector.

To define a matrix of size n rows by p columns:

```
<VARIABLE NAME> = [[ <value_1_1>, <value_1_2>,  , <value_1_p>;
<value_2_1>, <value_2_2>, , <value_2_p>;

<value_n_1>, <value_n_2>,, <value_n_p>]]
```

Note the double square brackets to define the matrix type, comma separated values to signify individual elements and semi-colon to specify the end of a row.

Alternatively to create a matrix, it is possible to use functions diagonal, triangle, matrix. These take a vector as input and return a matrix.

For example (UseCase6_2.mdl):

```
VARIABILITY {
 PPV_CL_V_KA : {matrixValue = triangle([0.1,
 0.01, 0.1,
 0.01, 0.01, 0.1], 3, true)}

 } # end VARIABILITY
```

Parameter naming

Unlike some target software, MDL does not have reserved names for parameters, nor is any meaning extracted from parameter names.

In the MDL documentation, we have used the convention that variability parameters describing the population parameter variability from the combination of between subject and within subject (between occasion) random effects are named PPV_. The individual level random effects we've named ETA_ since this is a familiar convention for many analysts. The residual unexplained variability parameters have been named RUV_ and the random variable associated with these has been named EPS_ again to following a familiar convention.

Covariances and Correlations

Random variability parameters and any covariances or correlations are defined separately, rather than as a combined matrix.

The covariance (or correlation) between random effects is defined as follows:

```
 <PARAMETER NAME> : { parameter = <vector of random effect
                      variables> ,
                      value = <vector of values>,
                      type is <cov | corr> }
```

The random effects variables must be declared in the `DECLARED_VARIABLES` block within the Parameter Object so they can be mapped to the random effect variables in the Model Object.

So for a simple example where the between subject variance parameters for CL, V and KA are on the standard deviation scale and the correlation between these parameters is to be specified, the standard deviation - correlation matrix (standard deviation on the diagonal, correlation off diagonal) is given by

$$\begin{bmatrix} PPV\_CL \\ PPV\_V \\ PPV\_KA \end{bmatrix} = \begin{bmatrix} sd = \ 0.1 & 0.01 & 0.01 \\ \mathbf{corr = 0.01} & sd = \ 0.1 & 0.01 \\ \mathbf{corr = 0.01} & \mathbf{corr = 0.01} & sd = \ 0.1 \end{bmatrix}$$

And the corresponding MDL code is:

```
warfarin_PK_CORR_par = parObj {
 DECLARED_VARIABLES{ ETA_CL ETA_V ETA_KA}
 STRUCTURAL {

 } # end STRUCTURAL

 VARIABILITY {
 PPV_CL : {value=0.1, fix=true, type is sd}
 PPV_V : { value = 0.1, type is sd }
 PPV_KA : { value = 0.1, type is sd }
 PPV_TLAG : { value = 0.1, type is sd, fix=true }
 # correlation between CL, V, KA
 OMEGA1 : {type is corr, parameter=[ETA_CL, ETA_V, ETA_KA],
           value=[0.01, 0.01, 0.01]}
 } # end VARIABILITY
 } # end of parameter object
```

In the code above, the variable OMEGA1 is defined as the lower triangle of the matrix above (correlation entries only) and three values are required to define the correlations between the parameters. Specifying the between subject variability parameters separately from covariances and correlations allows the user to change the covariance or correlation structure independently of the other variance parameter definitions.

Note that the parameters correlated are the random effects rather than the parameters defining the distribution of the random effects. Thus it is these random effect variables that are declared in the `DECLARED_VARIABLES` block.

Chapter 4

Model Object

The Model Object within the MDL is intended to describe the mathematical and statistical properties of the model. MDL defines language elements that allow the user to code a wide variety of models and in a variety of ways. The Model Object is intended to specify the model independent of the target software which will be used for the task (estimation or simulation). The same model should be able to be used for a variety of tasks – estimation, simulation or optimal design without recoding. The Model Object should also be independent of the data – where possible we use enumerated types for categorical covariates and outcomes so that definition of the model is clear to any user regardless of the data used in a given task.

It should be noted however that MDL does not guide the user about whether the model that is defined is suitable for a given purpose or for any target software. The user is free to define any model, however they must also be aware that the specified model may not be useable with all target software.

As stated in the Introduction, the Model Object is intended to convey the mathematical and statistical definitions required to completely define the model. MDL used in defining the model is intended more as a descriptive language rather than a programmatic one. The Model Object is used in tasks by combining it with Data, Parameter and Task Properties objects, and defining tasks within the R script.

Currently defined blocks are `IDV`, `COVARIATES`, `POPULATION_PARAMETERS`, `FUNCTIONS`, `VARIABILITY_LEVELS`, `STRUCTURAL_PARAMETERS`, `VARIABILITY_PARAMETERS`, `GROUP_VARIABLES`, `RANDOM_VARIABLE_DEFINITION`, `INDIVIDUAL_VARIABLES` , `MODEL_PREDICTION`, `DEQ`, `COMPARTMENT`, `OBSERVATION`.

Which blocks within the Model Object are used for a particular model depends on the structure of that model. Blocks should not be left empty (although this is not a syntax error). It is good practice to structure and write the model to facilitate readability and understanding of the model. Simple statements that are clear and unambiguous are preferred to statements combining many actions into one line of code. Use of the `MODEL_PREDICTION` block is encouraged to make it clear what the final prediction is from the model prior to use in generation of the observation level.

In the current version of MDL, the variable names and parameter names in the `STRUCTURAL_PARAMETERS` and `VARIABILITY_PARAMETERS` blocks of the Model Object must be matched to those in the Data and Parameter objects. The MOG Object brings together the Data, Parameter, Model and Task Properties Objects to perform tasks, and at this stage it is assumed that the variable names match across objects.

The independence of the Data Object from the model means that the data referenced by the Data Object may be easily used with a different model without modification of the Data Object. Similarly, the independence of the Parameter Object from the model means that all the parameters related to modelling project e.g. describing a particular drug, may be stored in one place. Note that parameters defined in the Parameter Object must have unique names.

Unlike other MDL Objects, the Model Object does not use a `DECLARED_VARIABLES` block. Instead variables are declared when they are used within the `IDV`, `COVARIATES`, `STRUCTURAL_PARAMETERS`, `VARIABILITY_PARAMETERS` blocks. Particular care may be required for models defined using analytic equations (rather than via differential equations, compartments). In these models it may be necessary to declare inputs such as DOSE within the `MODEL_PREDICTION` block.

On interoperability

The primary focus of MDL in this release is translation to valid PharmML, rather than conversion to target software. The previous Public Release was primarily concerned with demonstrating interoperability across key software targets. In this version of MDL there may be features supported which are not supported by certain target software, but which are valid for model description and which generate valid PharmML. The aim is to widen the scope of models which can be encoded in MDL and generate PharmML, since the latter is required for uploading models to the DDMoRe repository. Translation of these models to target software will follow with updates to the interoperability framework converters.

The MDL-IDE should assist the user in ensuring that the models encoded are valid MDL (and as a consequence, also valid PharmML).

Models in MDL may be expressed in a number of ways, which may be influenced by a number of factors including which languages the user is familiar with for encoding models. Flexibility allows the user to encode models quickly in a common language (MDL) which can then be shared with others and mutually understood. This flexibility also facilitates encoding in a given target when that language construct does not have a parallel in other tools. **However**, we **STRONGLY** encourage the user to encode the majority of models in a way that will facilitate interoperability. Interoperability allows the user of the model to choose the best tool for the job, or at least the tools that they have available to them.

If the user follows certain conventions for coding then it will increase the chance that a given model is interoperable between target tools. These conventions will be highlighted in the subsequent sections, but users should pay particular attention to sections 4.7, 4.9 and 4.10 on definition of `GROUP_VARIABLES` defining fixed effects, `INDIVIDUAL_VARIABLES` defining the relationship between covariates (or `GROUP_VARIABLES` defined variables) and random effects and `MODEL_PREDICTION` using these parameters to calculate predictions for given inputs.

IDV

The `IDV` block defines the independent variable within the model. Typically this is TIME (or T for differential equations). An `IDV` block must be present in the Model Object.

The syntax is a simple variable declaration:

```
IDV{ <Independent variable name> }
```

COVARIATES

The `COVARIATES` block declares and defines covariates to be used in the `GROUP_VARIABLES`, `INDIVIDUAL_VARIABLES` and `MODEL_PREDICTION` blocks (see discussion of regressors below for use of covariates in the `MODEL_PREDICTION` block). Covariates listed in the `COVARIATES` block must be specified as `use is covariate` or `use is catCov` in the `DATA_INPUT_VARIABLES` block in the Data Object or defined in the `POPULATION` block of the Design Object. Covariate transformations may be specified within this block.

```
COVARIATES{
< Covariate name >
< Categorical covariate name > withCategories {< category1 >, < category2 >, ... , < category_k >}
< Covariate name > = <simple transformation equation>
}
```

For categorical covariates, the categories defined in the `COVARIATES` block must match those specified for `DATA_INPUT_VARIABLES` with `use is catCov` – see the definition of the SEX covariate above.

An example `COVARIATES` block is shown below:

```
COVARIATES{
  WT
  SEX withCategories {female, male}
  logtWT = ln(WT/70)
}
```

In this example the withCategories prefix to the list of category names will be used to link to the values associated with these names in the Data Object.

The logtWT variable in the `COVARIATES` block may be used as the value for the cov attribute in the linear function in the `INDIVIDUAL_VARIABLES` block if `WT` follows the above rules for covariates.

Please also read about the specification of covariate models in sections 4.7 and 4.9.

The definition of covariates above assumes that the covariates are constant within individuals or vary only at occasion levels. It is also possible to define covariates that vary with the independent variable (typically time):

```
COVARIATES(type is idvDependent){
  WT
  logtWT = ln(WT/70)
}
```

Covariates which are defined as `idvDependent` should NOT be used in the `type is linear` definition of `INDIVIDUAL_PARAMETERS`.

STRUCTURAL_PARAMETERS

This block declares fixed effect parameters that define the structure of the model. There is no separator character in between variable names. The variable names do not need to be on separate lines, but it may be easier to read if they are presented in this way and it allows comments to be added to help communication

```
STRUCTURAL_PARAMETERS{
  < Variable name(s) of structural parameters >
}
```

For example:

```
STRUCTURAL_PARAMETERS {
  POP_CL
  POP_V
  POP_KA
  POP_TLAG
  BETA_CL_WT
  BETA_V_WT
} # end STRUCTURAL_PARAMETERS
```

VARIABILITY_PARAMETERS

Similar to the STRUCTURAL_PARAMETERS block, this block declares all the variability (including covariance, correlation and residual error) parameters (population parameter variability and other variability level parameters) used in the model. The variable names do not need to be on separate lines, but it may be easier to read if they are presented in this way and allows comments to be added to help communication

```
VARIABILITY_PARAMETERS{
<Variable name(s) of variability parameters>
}
```

For example:

```
VARIABILITY_PARAMETERS {
  PPV_CL
  PPV_V
  CORR_CL_V
  PPV_KA
  PPV_TLAG
  RUV_ADD
  RUV_PROP
} # end VARIABILITY_PARAMETERS
```

Residual Unexplained Variability

Residual variability is typically defined as a standard Normal distribution ~N(mean=0, var=1). The standard residual error models (see section 4.12.1) then define the parameters of that model e.g. additive and proportional which multiply the random N(0,1) variable. They can also define expressions involving parameters which define the residual error model. These parameters should be declared as VARIABILITY_PARAMETERS.

VARIABILITY_LEVELS

The VARIABILITY_LEVELS block defines the model hierarchy. Each variable should have attributes defining its level in the model hierarchy and variability type which is one of parameter or observation. DATA_INPUT_VARIABLES with use is dv and use is id are automatically identified as describing variability levels. Additional variables can be used to define variability levels by defining these as use is varLevel in DATA_INPUT_VARIABLES.

Typically, level = 1 is the level of each sample / experimental unit / observation. Additional levels of the hierarchy built on top of this. Typically in population models there is at least one additional level of variability – that of the individual (the experimental unit). Occasionally if modelling summary level data in a model-based meta-analysis, treatment arm may be used as the experimental unit and labelled in the DATA_INPUT_VARIABLES block as use is id.

The syntax is:

```
  VARIABILITY_LEVELS(reference = < ID | varLevel >){
      <Variable name> : { level = <number>, type is <parameter | observation>}
      ... # Additional levels of variability specified as above
  }
```

For example:

```
VARIABILITY_LEVELS(reference=ID){
  ID : { level=2, type is parameter }
  DV : { level=1, type is observation }
  }
```

If between occasion variability is required in the model then this should be specified here as a variability level between the observation and individual levels. In NONMEM occasion is typically specified as an additional layer of inter-individual variability which is defined conditionally on an occasion variable in the dataset. In MDL this is explicitly treated as a distinct level of variability.

```
VARIABILITY_LEVELS(reference=ID){
 ID : { level=3, type is parameter }
 OCC : { level=2, type is parameter }
 DV : { level=1, type is observation }
 }
```

Additional levels of variability are easily implemented by incrementing `level = <number>` with an associated DATA_INPUT_VARIABLE with `use is varLevel`. This facilitates definition of levels such as between trial random variability.

The distinction between `type is observation` and `type is parameter` will be used further in future versions of MDL to describe models where there are additional levels of hierarchy. For example, when describing differences between trials as a random effect or modelling population level differences; and at the observation level e.g. replicates of PD measurements at each time point, or multiple assays of a single sample. Specifying variability type allows extensibility for the future while retaining backwards compatibility.

When estimating model parameters using observed data, the `DATA_INPUT_VARIABLE` with `use is id` is the default “reference level of the model hierarchy (Gelman 2006). When using a Design Object, the reference level of the model hierarchy is likely to be implicit (subjects in a study arm) rather than explicit. Thus the need to specify `reference = ID`.

GROUP_VARIABLES

The `GROUP_VARIABLES` block can be used to specify group specific variables using parameters and fixed effect relationships between parameters and covariates. The `INDIVIDUAL_VARIABLES` block can then use these values in definition of the individual parameters by incorporating the random between individual variabilities defined in the `RANDOM_VARIABLE_DEFINITION` block(s).

Using the `GROUP_VARIABLES` block to define covariate relationships is not supported for parameter estimation in some target software since the equations defined in the `GROUP_VARIABLES` block are user defined. The MDL-IDE is not equipped to determine whether the defined relationships conform to linear relationships (after transformation) that have been shown to allow interoperability between software.

For this reason we suggest that definition of covariate dependent `GROUP_VARIABLES` is used only in cases where a reformulation to “linear or “linear after transformation relationships with covariates as defined in section 4.9 is not possible.

The `GROUP_VARIABLES` block is essential for defining relationships between structural parameters and covariates which are non-linear, even after transformation. For example to describe clearance across both adults and children a maturation model may be required. For example:

```
GROUP_VARIABLES{
  FSIZE = (WT/70)^0.75
  FAGE = if(AGE >= 20) then exp(BETA_CL_AGE*(AGE-20))
         else 1
  FMAT = 1/(1+(PCA/TM50)^(-HILL))
  GRP_CL = POP_CL * FSIZE * FAGE * FMAT
}
```

GRP_CL can then be used in the definition of individual variables within the `INDIVIDUAL_VARIABLES` block.

Defining model constants

Model constants (model variables with constant values) may be defined in MDL within the `GROUP_VARIABLES` block.

However, to ensure interoperability within the current SEE, constant values in the model should be defined as `STUCTURAL_PARAMETERS` and fixed to a value in the Parameter Object.

For models expressed as systems of differential equations (`DEQ` block), model variables can be set to constant values in the `MODEL_PREDICTION` block, but this may be computationally inefficient in the target software implementation.

RANDOM_VARIABLE_DEFINITION

The `RANDOM_VARIABLE_DEFINITION` block defines the distribution of the random effects to be used in construction of mixed effects models. The `RANDOM_VARIABLE_DEFINITION` block defines random variables in terms of parametric distributions.

It is assumed that all variables within the same block are defined for the same level of the model hierarchy. Separate `RANDOM_VARIABLE_DEFINITION` blocks should be used for each layer of the model hierarchy.The user specifies which level through the `(level = <name of variable associated with this level> )` syntax following the `RANDOM_VARIABLE_DEFINITION` block name.

The following syntax is used to define random variables:

```
RANDOM_VARIABLE_DEFINITION( level = <VARIABILITY_LEVEL variable> ){
  <VARIABLE NAME > ~ <Distribution with arguments>
 }
```

The `RANDOM_VARIABLE_DEFINITION` block supports probability distributions as specified in the ProbOnto knowledge base (M. J. Swat, Grenon, and Wimalaratne 2016). Typically for definition of structural parameter and residual error random variability, Normal distributions will be used. The MDL distribution "Normal(. . .) maps to either ProbOnto Normal1 or Normal2 depending on the parameterisation:

MDL Name	Argument name	Argument Types	ProbOnto distribution
Normal	mean	Real	Normal1
	sd	Real	
Normal	mean	Real	Normal2
	var	Real	

This means that the user does not need to remember which ProbOnto distribution uses which parameterisation for this frequently used distribution.

An example of `RANDOM_VARIABLE_DEFINTION` for individual random effects is given below:

```
RANDOM_VARIABLE_DEFINITION( level = ID ) {
  ETA_CL  ~ Normal(mean = 0, sd = PPV_CL)
  ETA_V   ~ Normal(mean = 0, sd = PPV_V)
  ETA_KA  ~ Normal(mean = 0, sd = PPV_KA)
  ETA_TLAG ~ Normal(mean = 0, sd = PPV_TLAG)
} # end RANDOM_VARIABLE_DEFINITION
```

In the code above, `ETA_CL`, `ETA_V`, `ETA_KA` and `ETA_TLAG` vary with each new value of ID. These variables are normally distributed with mean = 0 and standard deviation defined by the variability parameters. The distribution can also be defined using variances.

In the example above, all random variability parameters are independent. To specify correlation or covariance between parameters, the user should specify either pairwise correlation or covariance between random variables or use a multivariate distribution. In contrast to the previous version of MDL where correlations and covariances were defined only in the Parameter Object, this version requires the user to specify correlations and covariances in the `RANDOM_VARIABLE_DEFINITION` block. This is to allow the Prior Object to define priors on parameters used in the Model Object.

The following syntax is used to define correlation or covariance:

```
:: {type is <correlation / covariance>,
  rv1 = <RANDOM_VARIABLE_DEFINITION variable>,
  rv2 = <RANDOM_VARIABLE_DEFINITION variable>,
  variable = <VARIABILITY_PARAMETERS parameter> }
```

Note the use of the “anonymous list using double colon “:: . This is used since we are assigning additional information to the variable defined in the `VARIABILITY_PARAMETERS` block.

For example (UseCase1):

```
:: {type is correlation, rv1=ETA_CL, rv2=ETA_V,
value=CORR_CL_V}
```

Alternatively, the user can specify multivariate distribution(s) for parameters to specify the joint distribution of multiple random variability parameters. To do this, the user must specify the type of parameters in the `STRUCTURAL_PARAMETERS` (if required) and `VARIABILITY_PARAMETERS` blocks. Typically for multivariate distributions, there may be a vector of mean values, and a matrix of correlations or covariances. We can then use these to define the multivariate distribution using ProbOnto definitions.

So if we assume that the random effects for CL, V and KA (ETA_CL, ETA_V and ETA_KA in the univariate case) come from a multivariate distribution then the distribution of the vector ETA_CL_V_KA is given as:

$$ETA\_CL\_V_{\mathrm{KA}} = \begin{bmatrix} ETA\_CL \\ ETA\_V \\ ETA\_KA \end{bmatrix} \sim MultivariateNormal1 \left(mean = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, covariance = PPV\_CL\_V\_KA \right)$$

where PPV_CL_V_KA is a covariance matrix defining the variances and covariances of the random effects:

$$PPV\_CL\_V\_KA \ = \begin{pmatrix} PPV\_CL & COV\_CL\_V & COV\_CL\_KA \\ COV\_CL\_V & PPV\_V & COV\_V\_KA \\ COV\_CL\_KA & COV\_V\_KA & PPV\_KA \end{pmatrix}$$

Note that the ProbOnto distribution MultivariateNormal1 uses mean and covariance, while MultivariateNormal2 uses mean and correlation.

For example (UseCase6_2):

```
VARIABILITY_PARAMETERS {
PPV_CL_V_KA::matrix
PPV_TLAG
RUV_PROP
RUV_ADD
} # end VARIABILITY_PARAMETERS
```

```
RANDOM_VARIABLE_DEFINITION(level=ID) {
ETA_CL_V_KA ~ MultivariateNormal1(mean = [0,0,0],
covarianceMatrix = PPV_CL_V_KA)
ETA_TLAG ~ Normal(mean = 0, var = PPV_TLAG)
} # end RANDOM_VARIABLE_DEFINITION
```

Similarly for the residual unexplained variability with mean 0 and a fixed variance of 1, we might have a `RANDOM_VARIABLE_DEFINITION` block as follows:

```
RANDOM_VARIABLE_DEFINITION(level=DV){
EPS_Y ~ Normal(mean = 0, var = 1)
}
```

To define between occasion variability we might have a `RANDOM_VARIABLE_DEFINITION` block as follows:

```
RANDOM_VARIABLE_DEFINITION(level=OCC){
  eta_BOV_CL  ~ Normal(mean=0, var=BOV_CL)
  eta_BOV_V   ~ Normal(mean=0, var=BOV_V)
  eta_BOV_KA  ~ Normal(mean=0, var=BOV_KA)
  eta_BOV_TLAG ~ Normal(mean=0, var=BOV_TLAG)
 }# end RANDOM_VARIABLE_DEFINITION
```

Note that in the above example blocks, ID, DV and OCC are declared as valid identifiers for the variability hierarchy through the `VARIABILITY_LEVELS` block assuming appropriate specification within the Data Object of `DATA_INPUT_VARIABLES` with `use is id`, `use is varLevel` and `use is dv` for ID, OCC and DV (respectively).

```
VARIABILITY_LEVELS{
  ID : { level=3, type is parameter }
  OCC : { level=2, type is parameter }
  DV : { level=1, type is observation }
  }
```

The `RANDOM_VARIABLE_DEFINITION` block can also be used to explicitly define the distribution of individual variables, however if this method is used, then an anonymous list must be specified in the `INDVIDUAL_VARIABLES` block to refer to random variable defined in this way. Parameters defined in this way cannot be used with `type is linear` or `type is general` in the `INDIVIDUAL_VARIABLES` block.

For example:

```
RANDOM_VARIABLE_DEFINITION(level=ID) {
  CL ~ LogNormal3(median = POP_CL, stdevLog = SD_LNCL)
  ...
  }

INDIVIDUAL_VARIABLES{
  :: {type is rv, variable=CL}
  ...
  }
```

Covariates can be included via definition in the `GROUP_VARIABLES` block, but this may limit interoperability and translation of the model to target software.

For example:

(See `GROUP_VARIABLES` block example above for definition of GRP_CL)

```
RANDOM_VARIABLE_DEFINITION(level=ID) {
  CL ~ LogNormal3(median = GRP_CL, stdevLog = SD_LNCL)
  ...
}
```

The `RANDOM_VARIABLE_DEFINITION` block is also used to specify non-continuous outcome variables i.e. count, binary, categorical. See section 4.12.2.

INDIVIDUAL_VARIABLES

The `INDIVIDUAL_VARIABLES` block is used to express how the fixed effect variables (population parameters, covariates with their associated fixed effect parameters) and random effects (defined in the `RANDOM_VARIABLE_DEFINITION` block) combine to define the individual variables which will be used in the `MODEL_PREDICTION` block to calculate predictions for given inputs. If this is not a population model or if variables are completely defined through the `GROUP_VARIABLES` block then this block is not required. However, that might break interoperability with some tools like Monolix, which require the definition of individual parameters

There are three principle ways of defining `INDIVIDUAL_VARIABLES` and these will be described below.

The only way of defining `INDIVIDUAL_VARIABLES` that is currently supported for parameter estimation across target software is the “linear after transformation method described in section 4.9.1 below.

Mixed effect model with linear fixed effects and normally distributed

random effects

In some cases it is possible to express the fixed effects of covariates for a population parameter as a linear model with normally distributed random effects, sometimes employing a simple transformation (log, logit etc.) to achieve this.

We refer to this as a linear covariate model and this equates to the following mathematical definition:

$$h(\psi_i) = h\,(\ \psi_{\text{pop}}\ ) + \ \beta C_i + \eta_i$$

ψ_i – Individual parameter

ψ_{pop} – Typical or population mean parameter

β – Fixed effects

C_i – Covariates

η_i – Random effect

h – Transformation function – typically log, logit, probit etc.

The MDL syntax for this form of specification is:

```
<Individual parameter>
: {type is linear, trans is <h>,
pop = <Population STRUCTURAL parameter>,
fixEff = [ {coeff = <Fixed Effect STRUCTURAL parameter for covariate>,
cov = <Covariate in COVARIATES block conforming to rules below>}
,
... <Additional coefficient and covariate pairs as above> ],
ranEff = [ RANDOM_VARIABLE_DEFINITION parameter(s) ] )
```

The `fixEff` and `trans` arguments are optional.

Note that the syntax allows conditional assignment of list types to the parameter. So if the model for a parameter varies according to a covariate or variable value, then this can be reflected in the condition applied.

Note that left hand side transformations of the individual parameters are no longer allowed. The transformation specified in the `trans` argument applies to both the left hand side and right hand side of the equation. The ranEff argument expects a vector of random variables. If there is only one random variable the square brackets are not required.

For example (UseCase1):

```
CL : {type is linear, trans is ln, pop = POP_CL,
 fixEff = [{coeff=BETA_CL_WT, cov=logtWT}] ,
 ranEff = ETA_CL }
```

Using this construct for individual variables equates to the MU referencing approach in NONMEM and the standard definition of individual parameters in Monolix.

As discussed in the documentation of the Data Object defining covariates (section 3.3.5.1) certain constraints are placed on the type of covariate used in this form of specification. When covariates are defined within the Model Object `COVARIATES` block and used in the specification of `INDIVIDUAL_PARAMETERS` using the `linear( ..., fixEff=[{coeff=<coefficient>, cov = <covariate>}] )` construct, they must have particular properties:

- They must be constant within an individual or constant within an occasion.
- They will only be allowed simple transformations within the model e.g. centering on a median / mean and/or log transformation, logit transformation.
- The transformation cannot depend on another covariate.
- Statistical models (random effects) on covariates are not supported.

If a categorical covariate is used, then the catCov argument in fixEff should refer to the appropriate category of the covariate. For example (UseCase5):

```
CL : {type is linear, trans is ln,
  pop = POP_CL,
  fixEff = [
    {coeff = BETA_CL_WT, cov = logtWT},
    {coeff = POP_FCL_FEM, catCov = SEX.female },
    {coeff = BETA_CL_AGE, cov = tAGE}
    ],
  ranEff = ETA_CL }
```

If the categorical covariate has more than k>2 categories then the user needs to specify k-1 dichotomous “dummy covariates to specify the factor levels and appropriate contrasts (between level k and an appropriate comparison value). For example when adding GENOTYPE as a categorical covariate, the user may want to compare each category of GENOTYPE to a suitable reference value of the covariate. In this case all of the “dummy dichotomous comparison covariates should be included in the model in one step.

If between occasion variability is specified in a `RANDOM_VARIABLE_DEFINITION` block then the associated random effects can be specified in a vector form of the ranEff attribute. These will be added into the linear equation. For example (UseCase8):

```
CL : {type is linear, trans is ln, pop = POP_CL,
      fixEff = [{coeff=BETA_CL_WT, cov=logtWT}] ,
      ranEff = [eta_BSV_CL, eta_BOV_CL ]}
```

General mixed effect model with Gaussian random effects.

The second formulation for the `INDVIDUAL_PARAMETERS` block uses variables defined in the `GROUP_VARIABLES` block and assumes that the random effect is additive i.e. is Gaussian (Normally distributed) or Gaussian after transformation.

We refer to this as a "general or Gaussian after transformation model and the associated mathematical representation is:

$$h(\psi_i) = H\left(\beta, C_i\right) + \ \eta_i$$

ψ_i – Individual parameter

ψ_{pop} – Typical or population mean parameter

β – Fixed effects

C_i – Covariates

η_i – Random effect

H – Arbitrary function

h – Transformation function – log, logit, probit.

Where $H\left(\beta, C_i\right)$ is defined in the `GROUP_VARIABLES` block.

The MDL syntax for this form of specification is:

```
<Individual parameter> : {type is general,
                          grp = <GROUP_VARIABLES defined variable >,
                          trans is <ln / logit / probit>,
                          ranEff = [ RANDOM_VARIABLE_DEFINITION parameter(s) ] )
```

The trans argument is optional.

If the trans argument is used, then it is assumed that appropriate transformations have been made in the `GROUP_VARIABLES` block or in the assigned value for the grp attribute to ensure that the fixed effect and random effect are additive and on the correct scale given the transformation.

For example, for the GROUP_VARIABLES defined in section 4.7 above:

```
CL : {type is general, grp = ln(GRP_CL),
      trans is ln,
      ranEff = ETA_CL)
```

This corresponds to the following equation for CL:

$\ln\left(\text{CL}\right) = \ln\left(\text{GR}P_{\text{CL}}\right) + ETA_{\text{CL}}$

which, after back-transformation is equivalent to:

$CL = GRP_{\text{CL}} * exp(ETA_{\text{CL}})$

Mixed effect model defined by equations

The individual variables can also be defined using expressions by combining parameters with variables defined in `GROUP_VARIABLES` and random effects .

For example:

```
CL = POP_CL * exp(ETA_CL)
```

Or (using a variable GRP_CL defined in the `GROUP_VARIABLES` block as defined above)

```
CL = GRP_CL * exp(ETA_CL)
```

It is also possible to define fixed and random effect expressions as follows:

```
CL=POP_CL * (WT/70)^0.75 * exp(eta_PPV_CL)
```

Which can be log transformed into a linear form of mixed effect model like this:

```
CL=exp(ln(POP_CL)+ 0.75 * ln(WT/70) + eta_PPV_CL)
```

However, note that while it is possible for the user to "see that the equation above is linear in the fixed and random effects, it is not possible for the MDL-IDE to determine this. To specify linear models we must explicitly do so using the `{type is linear, ... }` construct described in section 4.9.1.

INDIVIDUAL_VARIABLES without inter-individual variability

For interoperability reasons, parameters defined in the `STRUCTURAL_PARAMETERS` block without associated variability i.e. where the individual value is the same as the parameter value, should be defined within the `INDIVIDUAL_VARIABLES` block. If the model parameter is constrained to be positive, then an appropriate transformation should be used to ensure a positive value.

`INDIVIDUAL_VARIABLES` where the variable is defined in the

`RANDOM_VARIABLE_DEFINITION` block.

As discussed in section 0 above, if the individual variable is defined completely via a distribution in the `RANDOM_VARIABLE_DEFINITION` block, then an anonymous list must be used to declare the variable within the `INDIVIDUAL_VARIABLES` block.

The syntax for the anonymous list is:

```
:: {type is rv, variable = <RANDOM_VARIABLE_DEFINITION
variable>}
```

Conditional assignment of `INDIVIDUAL_VARIABLES`

It is possible to apply conditional handling to the assignment of `INDIVIDUAL_VARIABLES`. Note that the conditioning occurs on the RIGHT HAND SIDE of the expression ONLY. The conditioning statement should follow the conventions described in section 9.1.4.4.

The syntax is as follows:

```
<Individual variable>
  : if(condition1) then { INDIVIDUAL_VARIABLE list 1 }
    elseif(condition2) then { INDIVIDUAL_VARIABLE list 2 }
    else { INDIVIDUAL_VARIABLE list 3 }
```

Note the use of an "else statement to ensure that is always assigned a value.

For example:

```
CL : if(RF==RF.normal) {type is linear, trans is ln,
                        pop = POP_CL, fixEff = {coeff = BETA_CL_WT, cov = logtWT},
                        ranEff = ETA_CL }

      else {type is general, trans is ln,
            grp = ln(GRP_CL),
            ranEff = ETA_CL }
```

In the above, the expression for individual Clearance is conditional on whether the subject's renal function (RF) is "normal or not. The user would need to have defined a suitable model for GRP_CL in the `GROUP_VARIABLES` block. (RF is a categorical data variable that has a symbolic value of "normal and other categories which are not used in this example).

Please also see sections 9.1.4.4 and 9.1.4.5 for further information on handling of conditional statements.

`INDIVIDUAL_VARIABLES` definitions in practice.

As has been discussed above, to facilitate interoperability we strongly suggest that users try to formulate their models using the {type is linear, ... } form shown in section 4.9.1 with the caveat included about the forms of covariate relationships that can be used within this construct.

In some cases, users may have to consider how their model is constructed more carefully. For example, in a pharmacodynamic model:

`PD = PD_BASELINE + PD_BETA\*CP + ETA_PD`

It may be tempting to try to write this as a `{type is linear, ... }` relationship with CP as a covariate, but recall that covariates may not be time-varying, and CP would almost certainly break this rule.

If we encode `GRP_PD = POP_BASELINE + POP_BETA\*CP` as a `GROUP_VARIABLE` and then add ETA_PD in `INDIVIDUAL_VARIABLES` using the `{type is general, ... }` form then the GRP_PD is also time-varying.

`INDIV_PD : {type is linear, pop = GRP_PD, ranEff = [ETA_PD])`

However if we break the above model into components, then we can use `{type is linear, ... }` to express an individual baseline

`INDIV_BASE : {type is linear, pop = POP_BASELINE, ranEff = ETA_BASE)`

`INDIV_BETA : {type is linear, pop = POP_BETA, ranEff = ETA_BETA)`

We can then move the linear relationship with CP to the `MODEL_PREDICTION` block

```
MODEL_PREDICTION{
  PD = INDIV_BASE + INDIV_BETA\*CP
  }
```

Using the `INDIVIDUAL_VARIABLES` block to define individual parameters which are then used in `MODEL_PREDICTION` should allow most models to be interoperable.

`MODEL_PREDICTION`

The `MODEL_PREDICTION` block is where the structural model predictions are defined. Calculations use mathematical expressions that may involve the population parameters (structural) as well as group and individual variables (parameters).

If a `MODEL_PREDICTION` block is not supplied this is not an error but requires that any prediction referred to in the `OBSERVATION` blocks has been defined using variables in a `GROUP_VARIABLES` or INDIVIDUAL_VARIABLE block.

For example below we present the `MODEL_PREDICTION` block using variables DOSE, V, CL, V and TIME.

```
 MODEL_PREDICTION{
 DOSE::dosingVar # recall that DOSE must be declared before use in
 analytical models.
 CONC=DOSE/V*exp(-CL/V*TIME)
 }
```

If a DEQ sub-block is specified then variables calculated within the DEQ sub-block can be referred to outside of this block to calculate the model prediction. An example is given below.

To ensure interoperability, any variable used in the `MODEL_PREDICTION` block must be either:

- the independent variable
- defined in `MODEL_PREDICTION`
- declared in INDIVIDUAL_VARIABLES using {type is linear, . . . }
- defined as "use is variable in the DATA_INPUT_VARIABLES block of the Data Object

This implies in particular that `STRUCTURAL_PARAMETERS`, `VARIABILITY_PARAMETERS`, `GROUP_VARIABLES` and random variables defined in `RANDOM_VARIABLES_DEFINITION` cannot be used in `MODEL_PREDICTION`.

DEQ

Use of a `DEQ` sub-block is optional – differential equations may be used anywhere in the `MODEL_PREDICTION` block – but it is encouraged to use this sub-block for clarity and readability of the resulting code.

The `DEQ` sub-block specifies the structural model through differential equations. The general form is

```
<VARIABLE> : { deriv = <expression>, init = <Real number>,
               x0 = <Real number> }
```

The `DEQ` sub-block combines equations and differential equations and the resulting system of equations is integrated across the independent variable, usually time.

`init = <Real number>` is the initial value of the differential equation

`x0 = <Real number>` is the starting value of the integrator. For most systems involving time, this is zero.

By default, `init = 0` and `x0 = 0`. If the default is to be used, these arguments can be dropped from specification of the differential equation.

For example:

```
MODEL_PREDICTION {
DEQ{
 RATEIN = if(T >= TLAG) then GUT \* KA
 else 0
 GUT : { deriv =(- RATEIN), init = 0, x0 = 0 }
 CENTRAL : { deriv =(RATEIN - CL \* CENTRAL / V) }
 }
 CC = CENTRAL / V
 } # end MODEL_PREDICTION
```

On Tlag and Bioavailability

Since MDL has no reserved variable names, there is no mechanism for target software to identify lagtime and bioavailability. . Models with lag times and bioavailability must use the `COMPARTMENT` sub-block to specify these input attributes.

COMPARTMENT

The `COMPARTMENT` sub-block is intended to provide the user with a modular approach to describe PK processes through definition of the drug input, distribution, and elimination processes. The functions defined are influenced by the PK macros approach in Monolix. The table below shows how the Compartment definitions in MDL correspond to PK Macros as defined in Monolix.

MDL Compartment	**Monolix PK Macros**
direct	iv
depot	absorption
elimination	elimination
distribution	peripheral
effect	effect
transfer	transfer
compartment	compartment

The major differences over the implementation in Monolix are that in MDL, the “from and “to attributes define the links between compartments and processes and that there are no reserved names for compartments or variables e.g. using “K12 and “K21 as variable names confers no special meaning to the use of these variables.

`COMPARTMENT` definitions are translated into PK Macros in Monolix and where possible they are mapped to ADVAN closed-form solutions in NONMEM.

`COMPARTMENT` sub-block processes are specified as lists with attributes depending on the processes being described.

Input & absorption

There are two `COMPARTMENT` block processes describing input to the system (typically drug input). These are direct and depot. direct defines bolus or zero-order input processes, while depot describes first-order, zero-order or transit chain drug input processes.

Input format is of the form:

```
<VARIABLE NAME> : { type is <depot / direct>,
                    to = <VARIABLE>,
                    <other arguments> }
```

The other arguments depend on the process being described. The table below describes the possible combinations of attributes for different input and absorption processes.

Compartment Type	**Attribute Combination**
Direct	to, modelDur(O), tlag(O), finput(O)
Depot	to, ka, tlag(O), finput(O)
	to, modelDur, tlag(O), finput(O)
	to, ka, ktr, mtt
	to, modelDur, ktr, mtt

(O) = Optional attribute

```
INPUT_KA : {type is depot, to=CENTRAL, ka=KA,
          tlag=ALAG1, finput=F1}
```

Distribution processes

MDL defines drug distribution (movement of drug between compartments) through `COMPARTMENT` block definitions with type compartment, distribution.

`type is compartment` is used to define PK compartments which have an associated input and elimination process. Peripheral compartments with `type is distribution` will not have `type is input` or `type is elimination` processes associated with them.

The syntax is:

```
<VARIABLE NAME> : { type is compartment }
```

The modelCmt argument is not used.

For example:

```
CENTRAL : { type is compartment }
```

Compartments where the transfer of drug in and out is defined through model variables have `type is distribution`.

```
 <VARIABLE NAME> : {type is distribution, from = <VARIABLE NAME>,
                    kin = <VARIABLE NAME>, kout = <VARIABLE NAME> }
```

For example, to specify a peripheral compartment in a two compartment PK model:

```
PERIPHERAL : {type is distribution, from=CENTRAL, kin=Q/V2, kout=Q/V3}
```

A “type is effect process provides a means to describe the transfer of amounts from a given compartment to an effect compartment e.g. for use with PD models.

```
<VARIABLE NAME> : {type is effect, from = <VARIABLE> NAME>, keq = <VARIABLE NAME>}
```

Compartment Type	Attribute Combination
distribution	from, kin, kout
compartment	
effect	from, keq

Elimination and transfer processes

Elimination is defined via a list with “type is elimination and specification of the compartment from which drug is eliminated along with variable names for the volume of distribution in the compartment from which drug is eliminated and the micro constant or apparent clearance from the compartment.

If the amount of eliminated drug is not of interest, it is not necessary to name this process. If this is the case, an anonymous list must be used:

```
 :: { type is elimination, from = <VARIABLE NAME>, v = <VARIABLE NAME>,
        <k / cl> = <VARIABLE NAME> }
```

For example in the one compartment model:

```
:: {type is elimination, from=CENTRAL, v=V, cl=CL}
```

Note that the v argument refers to the volume of distribution in the compartment defined by the from argument.

A “type is transfer process has also been provided which defines the one-way transfer of drug amounts from one compartment to another.

```
<VARIABLE NAME> : {type is transfer, from = <VARIABLE NAME>,
                    to = <VARIABLE NAME>, kt = <VARIABLE NAME> }
```

For example: `:: {type is transfer, from=LATENT, to=CENTRAL, kt=K23}`

Compartment Type	Attribute Combination
elimination	from, v, k
	from, v, cl
	from, vm, km
transfer	from, to, kt

Combining COMPARTMENT and DEQ blocks

It is possible to use `COMPARTMENT` to describe the input processes for differential equations in the `DEQ` block. Using the `type is depot` or `type is direct` compartments allows the user to specify lag time and bioavailability (tlag and finput) which will translate to appropriate terms in target software e.g. ALAGn and Fn in NM-TRAN. If the `COMPARTMENT` specification is not used then model parameters are treated in a very general way and there is no way of mapping these to target tools to implement these input attributes.

If the `COMPARTMENT` sub-block is not used then delay absorption processes and/or bioavailability needs to be explicitly encoded.

For example in UseCase4 differential equations are used to describe IV and oral administration. The time of dosing DT is passed either from the data or via `DATA_DERIVED_VARIABLES` (see section 2.5):

```
MODEL_PREDICTION {
  DT
  DEQ{
    RATEIN = if(T-DT >= TLAG) then GUT * KA
    else 0
    GUT : { deriv =(- RATEIN), init = 0, x0 = 0 }
    CENTRAL : { deriv =(RATEIN * FORAL - CL * CENTRAL / V), init = 0, x0 = 0 }
}
CC = CENTRAL / V
} # end MODEL_PREDICTION
```

In the above example, note that there is a discontinuity in RATEIN which is not well handled with differential equations. Note also that the model does not handle multiple doses since then DT (time of dose) and hence RATEIN would be reset to zero for each new dose.

To alleviate this problem, we can use `COMPARTMENT` with `DEQ` to handle the input processes correctly. The code below shows how to include lag time and bioavailability in a model using differential equations (UseCase4_2):

```
MODEL_PREDICTION {
COMPARTMENT{
  INPUT_KA : {type is depot, to=CENTRAL, ka=KA, finput=FORAL, tlag=TLAG}
  INPUT_CENTRAL : {type is direct, to = CENTRAL}
}

DEQ{
  CENTRAL : { deriv =( - CL \* CENTRAL / V), init = 0, x0 = 0 }
  }
CC = CENTRAL / V
} # end MODEL_PREDICTION
```

(Note that this model is not exactly equivalent to UseCase4 which does not allow for superposition of dosing). Note the first-order input to the CENTRAL compartment (INPUT_KA) is `type is depot` while the bolus or zero-order rate input (INPUT_CENTRAL) is `type is direct`.

The model corresponding to UseCase4 is shown below (UseCase4_3). In this case there is an explicit differential equation for the depot compartment (GUT)

```
MODEL_PREDICTION {
COMPARTMENT{
INPUT_KA : {type is direct, to = GUT, finput=FORAL,
tlag=TLAG}
INPUT_CENTRAL : {type is direct, to = CENTRAL}
}

DEQ{
GUT : { deriv =(- GUT \* KA), init = 0, x0 = 0 }
CENTRAL : { deriv =(GUT \* KA - CL \* CENTRAL / V), init = 0,
x0 = 0 }
}
CC = CENTRAL / V
} # end MODEL_PREDICTION
```

In the code above, note that both administrations use `type is direct`, but the oral administration (INPUT_KA) has finput and tlag specified. In contrast to the above, the direct input would exactly correspond to the DEQ specification in UseCase4, however the TLAG in UseCase4 is treated as a general parameter, while the TLAG in UseCase4_3 will be translated to ALAGn in NMTRAN.

OBSERVATION

The `OBSERVATION` block provides the definition of the outcome variable using the prediction from the `MODEL_PREDICTION` block and/or `RANDOM_VARIABLE_DEFINITION` at the observation level. Calculations or equations needed for this definition can be defined in the `OBSERVATION` block e.g. calculation of weights for `type is userDefined` error models.

If more than one outcome is specified then we specify each outcome separately within the `OBSERVATION` block. Conditional assignment to outcome definitions is possible if the outcome depends on a covariate or calculated variable. However it is not necessary to conform multiple outcomes to a single observation variable name e.g. Y.

Continuous outcomes

For continuous outcomes the `OBSERVATION` block defines how a variable from the `MODEL_PREDICTION` block and `RANDOM_VARIABLE_DEFINITION` providing the residual unexplained random variables are combined in a function to define the outcome.

The mathematical representation of the outcome variable is (after Lavielle, 2014)

$$h\left(y_{\mathrm{ij}}\right) = h\left(f\left(x_{\mathrm{ij}}, \psi_i\right)\right) + g\left(f\left(x_{\mathrm{ij}}, \psi_i\right),\ \xi\right)\varepsilon_{\mathrm{ij}}$$

Where

$$y_{\mathrm{ij}} = \text{jth observation for subject } i$$

h = Transformation of the outcome to ensure that the resulting function is an additive function of f and g. Specfied in MDL error models described below using "trans is . Left hand side transformation is specified using the Boolean lhsTrans.

f = structural model prediction from the `MODEL_PREDICTION` block. Specified in the MDL error models described below through the "prediction argument.

g = functional definition of the residual error model. Specified in the MDL error models through the `type is additiveError | proportionalError | combinedError1 | combinedError2`

ψ_i = individual parameters defined in the INDIVIDUAL_VARIABLES block

$x_{i,j}$ = covariates and regression variables e.g. time, concentration etc.

ξ = parameters of the residual error model defined in the VARIABILITY_PARAMETERS block and referred to in the appropriate `additive` and `proportional` arguments.

$\varepsilon_{\mathrm{ij}}$= residual error defined in `RANDOM_VARIABLE_DEFINITION` block and referred to in the MDL error models described below through the `eps` argument.

The syntax for definition of continuous outcome variables is:

```
< OUTCOME VARIABLE NAME> : {type is additiveError |
proportionalError | combinedError1 | combinedError2 | userDefined,
<additional arguments defined in table below> }
```

The following residual error model functions are defined, as described in MDL Language Reference section 2.21 and reiterated here.

Name	**Return Type**	**Argument name**	**Argument Types**
additiveError	Real	trans (Optional)	Builtin
		lhsTrans	Boolean
		additive	Real
		prediction	Real
		eps	Real
proportionalError	Real	trans (Optional)	Builtin
		lhsTrans	Boolean
		proportional	Real
		prediction	Real
		eps	Real
combinedError1	Real	trans (Optional)	Builtin
		lhsTrans	Boolean
		additive	Real
		proportional	Real
		prediction	Real
		eps	Real
combinedError2	Real	trans (Optional)	Builtin
		lhsTrans	Boolean
		additive	Real
		proportional	Real
		prediction	Real
		eps	Real
userDefined	Real	value	Real
		weight	Real
		prediction	Real

`combinedError1` defines the following model:

$$h\left(y_{\mathrm{ij}}\right)=h\left(f\left(x_{\mathrm{ij}},\psi_i\right)\right)+\left(a+bf\left(x_{\mathrm{ij}},\psi_i\right)\right)\varepsilon_{\mathrm{ij}}$$

`combinedError2` defines the following model:

$$h\left(y_{\mathrm{ij}}\right)=h\left(f\left(x_{\mathrm{ij}},\psi_i\right)\right)+\sqrt{a^2+b^2f\left(x_{\mathrm{ij}},\psi_i\right)}\varepsilon_{\mathrm{ij}}$$

the lhsTrans Boolean argument allows the user to apply transformations to the DV without having to transform the data column prior to analysis.

As with the `type is linear` and `type is general` definitions in the `INDIVIDUAL_VARIABLES` block, we use defined types here to make explicit the relationships between predictions and residual random variable terms to facilitate interoperability between target software.

Use `type is userDefined` to specify an arbitrary relationship between prediction, the residual error random variable which is typically Normal(0,1), and the with associated function g(.) defined above. Using this form ensures that correct calculation of the weighted residuals can be calculated.

The current version of MDL does not support definition of outcomes with arbitrary functions of variables and random variables. Any equations written in the `OBSERVATION` block are treated as variables to be used in definition of the observation through a list definition as described above (including UserDefined). If a list definition is not used, then the observation equation may not be translated correctly to the target software tool.

Discrete data

Discrete data outcomes are described by referencing a suitable distribution for the outcome. In this version of MDL we assume that the parameters of the relevant distributions are supplied either in the data, for example the number of trials, N, in a binomial distribution, or are defined in the `MODEL_PREDICTION` block.

In this version of MDL we assume an identity link for all models – that is the parameter supplied to the distribution must be on the appropriate scale for that distribution – the Poisson rate parameter must have a positive value, probabilities for binary and categorical distributions must be on the scale (0,1).

Count, discrete, categorical outcomes must be specified within the `RANDOM_VARIABLE_DEFINITION` block for the dv level via a suitable ProbOnto distribution (see also section 0). The random variable is then declared in the `OBSERVATION` block via an anonymous list, as has been seen previously when defining individual variables via the `RANDOM_VARIABLE_DEFINITION` block. Since continuous outcomes have a residual error specified at the DV level, it is inferred that the outcome defined in the `OBSERVATION` block is at the DV level of variability. However for other types of data, it is less clear that this is the case. Thus we must use `RANDOM_VARIABLE_DEFINITION` to define these outcomes using ProbOnto definitions and then provide additional information in the `OBSERVATION` block to assign additional attributes.

The syntax is as follows:

```
RANDOM_VARIABLE_DEFINITION(level=DV){
  <outcome variable> ~ <ProbOnto distribution>
}

OBSERVATION{
  :: {type is <count/discrete/categorical>, variable = <outcome variable>}
}# end ESTIMATION
```

See below for examples pertaining to specific outcome types.

Count data

For count data, we have the following syntax:

```
RANDOM_VARIABLE_DEFINITION(level=DV){
<variable> ~ <ProbOnto distribution for count data e.g. Poisson1>
}

OBSERVATION{
:: {type is count, variable = <variable>}
}# end ESTIMATION
```

For example (also showing the appropriate `INDIVIDUAL_VARIABLES`, `MODEL_PREDICTION` and `OBSERVATION` blocks) (UseCase11)

```
INDIVIDUAL_VARIABLES{
BASECOUNT : {type is linear, trans is ln,
pop = POP_BASECOUNT, ranEff = eta_PPV_EVENT }
BETA = POP_BETA
}# end INDIVIDUAL_VARIABLES

MODEL_PREDICTION{
lnLAMBDA=ln(BASECOUNT) + BETA\*CP
LAMBDA = exp(lnLAMBDA)
}

RANDOM_VARIABLE_DEFINITION(level=DV){
Y ~ Poisson1(rate=LAMBDA)
}

OBSERVATION{
:: {type is count, variable = Y}
}# end ESTIMATION
```

Note in the above example that the BASECOUNT variable is specified using the linear function and a natural log transformation on both sides to ensure that BASECOUNT is positive. The linear relationship with CP (plasma concentration) is defined within the `MODEL_PREDICTION` block. We cannot use CP as a covariate in the `linear(...)` function as CP varies with time and so is regarded as a regressor rather than a covariate. In UseCase11 since there is no model for the pharmacokinetics we use CP as the independent variable (`IDV`) in the model and `use is idv` in the `DATA_INPUT_VARIABLES` block. We also take exponential of lnLAMBDA to ensure that the variable LAMBDA is on the positive scale before using this in the Poisson distribution.

This is an example where a little consideration of the random effects and model prediction can facilitate interoperability. Writing an equation for the INDVIDUAL_VARIABLES we may have defined

```
INDIVIDUAL_VARIABLES{
 lnLAMBDA = ln(POP_BASECOUNT) + BETA\*CP + eta_PPV_EVENT
 }
```

Using this formulation of the model though would not guarantee interoperability with some target software for estimation since the equation for lnLAMBDA is user-defined.

There are many distributions defined in ProbOnto which will describe count data. The diagram below (Figure @ref(fig:ProbOntoPoisson)) illustrates a few of these and relationships between them as described in the ProbOnto Knowledge Base (www.probonto.org)

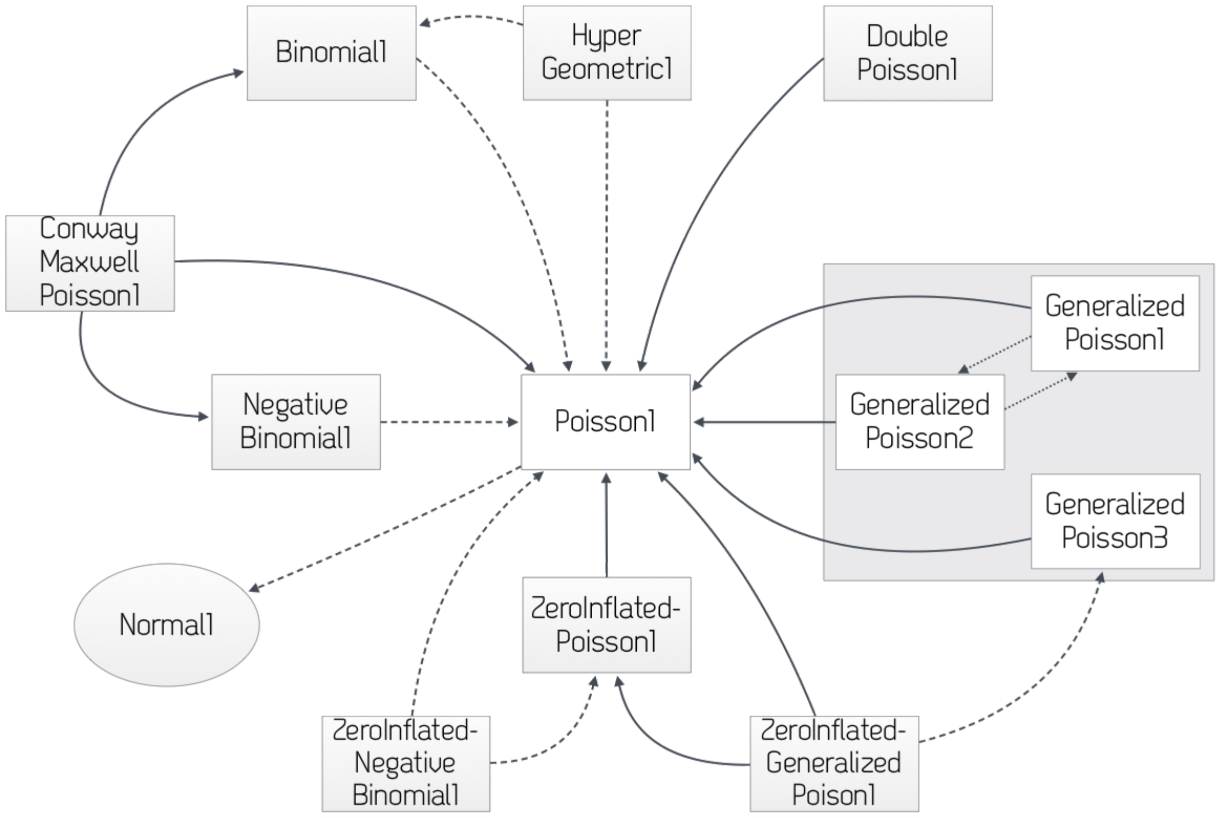

Figure 4.1: ProbOnto Poisson distributions

Binary data

Similar to count data above, we define the binary outcome and its distribution in a `RANDOM_VARIABLE_DEFINITION` block at the observation level of variability. Note how we define the names of the categories on the left hand side, and then the probability distribution defines the likelihood of the *second* category.

```
RANDOM_VARIABLE_DEFINITION(level=DV){
  Y withCategories {<category1>,<category2>}
     ~ <ProbOnto distribution e.g. Bernoulli | Binomial>
}
```

For example (again, showing the `INDIVIDUAL_VARIABLES`, `MODEL_PREDICTION` and `OBSERVATION` blocks to show the model construction):

```
RANDOM_VARIABLE_DEFINITION(level=ID){
  eta_PPV_EVENT ~ Normal(mean=0, var=PPV_EVENT )
}# end RANDOM_VARIABLE_DEFINITION

INDIVIDUAL_VARIABLES{
  indiv_BASE : {type is linear, pop= POP_BASEP,
  ranEff=[eta_PPV_EVENT], trans is logit}
}# end INDIVIDUAL_VARIABLES

MODEL_PREDICTION{
  LP = logit(indiv_BASE) + POP_BETA\*CP
  P1 = invLogit(LP)
}# end MODEL_PREDICTION

RANDOM_VARIABLE_DEFINITION(level=DV){
  Y withCategories {none, event} ~ Bernoulli1(probability=P1)
}

OBSERVATION{
  :: {type is discrete, variable = Y }
}# end ESTIMATION
```

Note that the `INDIVIDUAL_VARIABLES` block defines the individual baseline by combining the population parameter and the random effect. Note also that this specification uses a logit transformation to ensure that the individual baseline indiv_BASE variable is on the (0,1) probability scale. Then, the linear regression with plasma concentration (CP) is defined in the `MODEL_PREDICTION` – Note that CP is not a covariate. Finally LP is back-transformed to the probability scale to give variable P1 which is the probability of an event to be used in the Bernoulli distribution. By defining how the 0,1 in the data correspond to named events {none, event} it is easier to understand exactly what category is being modelled.

An alternative distribution for the same model is the Binomial distribution with one trial:

```
RANDOM_VARIABLE_DEFINITION(level=DV){
  Y withCategories {none, event} ~ Binomial1(numberOfTrials=1, probability=P1)
}

OBSERVATION{
  :: {type is discrete, variable = Y }
} # end of OBSERVATION
```

Categorical data

Again, similar to count and binary data, the syntax for Categorical data outcomes is:

```
RANDOM_VARIABLE_DEFINITION(level=DV){
<variable> withCategories{ <category1>, ..., <categoryk> }
          ~ <ProbOnto distribution for categorical data
          e.g. CategoricalNonordered1 | CategoricalOrdered1>
}

OBSERVATION{
  :: {type is categorical, variable=<variable>}
}
```

For example (UseCase13_1):

```
GROUP_VARIABLES{
  B0 = Lgt0
  B1 = B0 + Lgt1
  B2 = B1 + Lgt2
}

INDIVIDUAL_VARIABLES{
  indiv_B0 : {type is general, grp = B0, ranEff = eta_PPV_EVENT}
  indiv_B1 : {type is general, grp = B1, ranEff = eta_PPV_EVENT}
  indiv_B2 : {type is general, grp = B2, ranEff = eta_PPV_EVENT}
}# end INDIVIDUAL_VARIABLES

MODEL_PREDICTION{
  EDRUG = Beta * CP
  A0 = indiv_B0 + EDRUG
  A1 = indiv_B1 + EDRUG
  A2 = indiv_B2 + EDRUG
  P0 = invLogit(A0)
  P1 = invLogit(A1)
  P2 = invLogit(A2)
  Prob0 = P0
  Prob1 = P1 - P0
  Prob2 = P2 - P1
  Prob3 = 1 - P2
} # end MODEL_PREDICTION

RANDOM_VARIABLE_DEFINITION(level=DV){
  Y withCategories{ none, mild, moderate, severe }
      ~ CategoricalOrdered1(categoryProb=[Prob0, Prob1, Prob2, Prob3])
}

OBSERVATION{
  :: {type is categorical, variable=Y}
}
```

In the above code, the cutpoints between categories are defined in the `GROUP_VARIABLES` block (B0, B1, B2) and individual values for these are defined in the `INDIVIDUAL_VARIABLES` block. The linear effect of CP (plasma concentration) is defined in the `MODEL_PREDICTION` block and this is added to the individualised cutpoints (A0, A1, A2). These are then back-transformed to the probability scale (P0, P1, P2) and the ordered categorical model is defined by calculating the probability of each category as the difference from the previous category – Prob0, Prob1, Prob2, Prob3.

Time to event data

Time to event (TTE) models are modelled by specifying the hazard function. The PharmML to target software tool converters handle the translation of the hazard specification to target tool implementation. For some software this involves calculation of the survival function and associated likelihood.

For an arbitrary hazard function

$$\lambda(t)$$

:

Hazard function	$\lambda(t)$
Cumulative hazard function	$\Lambda(a,\ b) = \int_a^b \lambda(t)\,\mathrm{dt}$
Survival function	$P(T > t) = e^{-\Lambda(t_0,t)}$
Probability density function	$p(t) = \lambda(t)\,e^{-\Lambda(t_0,t)}$
Cumulative distribution function	$P(T < t) = \int_0^t p(s)\,\mathrm{ds}$

For an introduction to TTE models see (Holford 2013) and for a tutorial in implementation in NONMEM and Monolix see (N. H. M. Lavielle 2011).

The MDL syntax for time to event outcomes is :

```
<OUTCOME VARIABLE NAME> : { type is tte, hazard = <VARIABLE> }
```

For example (UseCase14.mdl):

```
INDIVIDUAL_VARIABLES{
  BTATRT = POP_BTATRT
  H_BASE = POP_HBASE
}

MODEL_PREDICTION{
  HBASE=H_BASE/365
  HAZTRT=BTATRT * TRT
  HAZ = HBASE * (1+HAZTRT)
} # end MODEL_PREDICTION
```

```
OBSERVATION{
  Y : {type is tte, hazard = HAZ }
} # end ESTIMATION
```

In the above case the hazard and the effect of treatment on the hazard is calculated in the `GROUP_VARIABLES` block. This is then used in the `MODEL_PREDICTION` block to calculate the hazard for the event. The model outcome variable Y is then defined as having type tte and the hazard calculated in the `MODEL_PREDICTION` block is passed in as an argument. Specification of the model is then very simple for the user – no calculation of Survival functions nor likelihood is necessary.

In the current MDL, TTE models are able to be handled equally by NONMEM and Monolix. To facilitate this, we impose certain constraints on dataset conventions. There must be a data record at the start of the interval during which the hazard will be integrated. We use `DV = 0` to denote right censoring and `DV = 1` to denote an event.

Currently only exact time of event and right censoring is supported in MDL. Future versions will support interval censoring and repeated time to event.

The convention in NONMEM datasets of using MDV to identify the start of the observation period for assessing TTE cannot be used.

FUNCTIONS

This block allows users to define their own functions, for example for use with interpolation.

The syntax is as follows:

```
<function name> :: function(<argument1> :: <argument1 type>,
    <argument2> :: <argument2 type>,
    <argumentk> :: <argumentk type>) :: <function result type>
    is
      <expression using argument1... argumentk>
```

Specifying the types of each argument and the type of the function result allows validation of the function inputs and outputs.

To call the function, the user types `&` before the function name.

For example:

```
FUNCTIONS{ myInterp::function(t::real, x0::real, t0::real, x1::real,
            t1::real)::real
            is
              x0
 }

DATA_INPUT_VARIABLES {
  ID : { use is id }
  TIME : { use is idv }
  WT : { use is covariate, interp=&constInterp }
  AGE : { use is covariate, interp=&myInterp }
  }
```

POPULATION_PARAMETERS

The POPULATION_PARAMETERS block allows the user to define models and model parameters that exist at the population level. This may be used in hierarchical models to define population parameters when there are additional levels of hieararchy above the individual.

In most "population approach models, the individuals are assumed to be drawn from a population, the characterstics of which are described through fixed and random effect models. Inferences are then made on the "population parameters which we take to be representative of the population from which the individuals are drawn.

In meta-analysis across many studies, regions, demographic populations it may be useful to characterise additional levels of hierarchy characterising how individuals within each study, region or demographic differ systematically (through fixed effect models) and randomly (through variability models). These higher level models can be expressed in the POPULATION_PARAMETERS block.

In the current MDL, population models are defined through combination of RANDOM_VARIABLE_DEFINITION and POPULATION_PARAMETERS definitions. In the current MDL expressions (equations) are NOT allowed in the POPULATION_PARAMETERS block.

The POPULATION_PARAMETERS block uses random variables defined in the RANDOM_VARIABLE_DEFINITION block. The syntax for the POPULATION_PARAMETERS block is as follows:

```
POPULATION_PARAMETERS{
  :: {type is < continuous | categorical >,
      variable = < RANDOM_VARIABLE_DEFINITION variable > }
}
```

For example (/FourModels/Hierarchical_Model.mdl):

```
RANDOM_VARIABLE_DEFINITION(level=POP){
  w_pop ~ Normal(mean = ws, sd = gw)
  V_pop ~ Normal(mean = Vs, sd = gV)
} # end RANDOM_VARIABLE_DEFINITION


POPULATION_PARAMETERS{
:: {type is continuous, variable=w_pop}
:: {type is continuous, variable=V_pop}
}

RANDOM_VARIABLE_DEFINITION(level=ID) {
  ETA_BSV_V ~ Normal(mean = 0, sd = omega_V)
  } # end RANDOM_VARIABLE_DEFINITION

INDIVIDUAL_VARIABLES {
  V : {type is linear, trans is ln, pop=V_pop,
       fixEff={coeff=BETA_WT, cov=WT}, ranEff=ETA_BSV_V}
  } # end INDIVIDUAL_VARIABLES

MODEL_PREDICTION {
  D
  f = D/V * exp(-k*T)
  } # end MODEL_PREDICTION
```

In the model above, the mean weight for each population is drawn from a Normal random variable. The mean Volume of distribution also varies for each population and is similarly drawn from a Normal random variable. The parameters for inference would be Vs ("global mean of Volume of distribution), gV (between population variability), omega_V (between individual variability), BETA_WT (fixed effect of Weight). V_pop gives the population prediction of Volume of distribution for each population observed in the data, while V gives the individual predicted Volume of distribution.

Chapter 5

Task Properties Object

The Task Properties Object is intended to convey information to specific target software related to algorithms, settings, options for performing a given task. Some Task Properties Object settings are generic (apply across different target software) but most are specific to the intended target software.

Task properties are specific to the task e.g. `ESTIMATE`, `SIMULATE`, `OPTIMISE`, `EVALUATE`. `TARGET_SETTINGS` are specific to a given target software tool. If the user intends to estimate with a given tool but no `ESTIMATE` block is given in the Task Properties Object and/or if the task properties are not specified for the intended tool, then default settings are used.

Intended use of Task Properties

The user should provide information about settings and options for each target software that they wish to use for a given (estimation) task. These settings and options might be specified to ensure reproducibility of results regardless of target software i.e. to ensure that the results from a given target software are comparable with results from different target software.

Alternatively, the user may wish to provide settings and options that they will frequently use with a given target software – so that every time they estimate with a given target they use the same settings.

The modularity of MDL allows the user to preset or reuse Task Properties objects between models. The user may then have preferred Task Properties for estimation that can be called upon during the relevant points in their M & S workflow.

As with other MDL blocks, it is possible to specify multiple Task Properties blocks within a single .mdl file and then reference only the one required for use with a specific task in either the MOG Object or via R. This allows the user to specify preferred settings for multiple target software tools – facilitating reproducibility and reuse. Even though many Task Properties Object can be given in an .mdl file, the MOG Object should contain only ONE Task Properties Object.

Why use Task Properties for settings and options rather than arguments of functions in the ddmore R package?

Task Properties are distinct from arguments to the ddmore R package functions for executing tasks – the former passes information to the appropriate target software about the particular settings and options required for a given task. The ddmore function arguments are equivalent to the command line settings or options which are employed when invoking target software.

For example, we may use Task Properties to define the estimation algorithm and associated settings with NONMEM, but define command line options for PsN which govern how NONMEM should be called by PsN.

How are Task Properties used by MDL and PharmML?

The Task Properties generic items are parsed and understood by the MDL editor within the MDL-IDE. However target software specific settings and options are not parsed by the MDL-IDE – these are passed as is via the PharmML to the target software converter where they are interpreted and converted where they may appear in target software code, external settings or options files as appropriate.

ESTIMATE

The syntax for this block is:

```
set algo is < foce | focei | saem | mcmc >
```

TARGET_SETTINGS

The `TARGET_SETTINGS` block holds specific settings for the given target software. These settings may be given in name – value pairs within the `TARGET_SETTINGS` block, or they may be specified in an external file.

MDL does not check whether the specified options within the `TARGET_SETTINGS` block are appropriate for a given tool. This is performed by the converter for the target tool. See the MDL Reference section for a list of which options are available for each target software.

The syntax for the `TARGET_SETTINGS` block is:

```
TARGET_SETTINGS(target="NONMEM" | "MONOLIX" | "BUGS" | "PFIM",
      settingsFile = "< filename >"){
set < target specific setting name > = < target specific setting value >,
    < target specific setting name > = < target specific setting value >,
    ...
  }
```

Note the comma separation between elements of the set statement.

For example (UseCase1_1):

```
SAEM_task = taskObj {
  ESTIMATE{
    set algo is saem
      TARGET_SETTINGS(target="NONMEM"){
      # Statements from the NONMEM control stream
      # $EST METHOD=SAEM INTERACTION NBURN=3000 NITER=500 PRINT=100 SEED=1556678 ISAMPLE=2
      # $COV MATRIX=R PRINT=E UNCONDITIONAL SIGL=12

      # Equivalent MDL statements
      set INTER=true, NBURN=3000, NITER=500, PRINT=100, SEED=1556678, ISAMPLE=2,
          COV_=true, COV_MATRIX="R", COV_PRINT="E", COV_UNCONDITIONAL=true, COV_SIGL=12
      } # end of TARGET_SETTINGS

      TARGET_SETTINGS(target="MONOLIX", settingsFile=["tables.xmlx"]){
          set graphicsSettings="tables.xmlx"
```

```
        } # end of TARGET_SETTINGS
   } # end of ESTIMATE statement
} # end of Task Object

BUGS_task = taskObj{
  ESTIMATE{
    set algo is mcmc
    TARGET_SETTINGS(target="BUGS"){
      set nchains = 1, # Number of MCMC chains
      burnin = 1000, # Number of MCMC Burn-in iterations
      niter = 5000, # Number of iterations
      parameters = "V,CL,KA,TLAG", # Parameters to monitor
      odesolver="LSODA", # or "RK45"
      winbugsgui ="false" # or "true"
      } # end of TARGET_SETTINGS
  } # end of ESTIMATE statement
} # end of Task Object

FOCEI_task= taskObj {
  ESTIMATE{
    set algo is focei
      TARGET_SETTINGS(target="NONMEM"){
      # Statements from NONMEM control stream
      # $EST METHOD=COND INTERACTION MAXEVAL=9999 NSIG=3 SIGL=10 PRINT=5 NOABORT NOPRIOR=1 FILE=example
      # $COV MATRIX=R PRINT=E UNCONDITIONAL SIGL=12

      # Equivalent MDL
      set INTER=true, MAXEVAL=9999, NSIG=3, SIGL=10, PRINT=5, NOABORT=true,
      NOPRIOR=1, FILE="example1.ext",
      COV_MATRIX="R", COV_PRINT="E", COV_UNCONDITIONAL=true, COV_SIGL=12,
      MSFO_FILE="MSFO.msf"
      } # end of TARGET_SETTINGS
    } # end of ESTIMATE statement
} # end of task object
```

For BUGS, the following properties are valid (as shown above):

```
nchains = <integer>
niter = <integer>
parameters = <comma-separated character string of parameter names>
odesolver = <"LSODA" | "RK45">
winbugsgui = <"true" | "false">
```

For NONMEM, the following rules are used in the `TARGET_SETTINGS` block:

- Individual properties influence only `$EST` statement or the `$COV`
- Properties starting with the string `COV_` will be added to the `$COV` statement using the following rules:
  - The string `COV_` will be removed from the property
  - If the remainder of the property is a string/integer assignment, it will be added as a parameter on the `$COV` statement verbatim (with any unnecessary quotes removed)
  - If the remainder of the property is a boolean assignment resolving to true, just the parameter name will be added on the `$COV` statement verbatim

- If there are any `COV_` properties present, then a `$COV` statement will be generated. Otherwise it will not be generated

- Any property starting with a string that is not `COV_` will be added to the `$EST` statement using the same string/integer/boolean rules as for `$COV`.

Chapter 6

Prior Object

For Bayesian estimation, the user must specify a Prior Object, which is used in place of the Parameter Object. The Prior Object defines prior distributions or values for all model parameters – both structural and variability. The prior distributions form an additional level of model hierarchy above all other levels of variability in the model. This level of the hierarchy does not need to be explicitly defined in the Model Object `VARIABILITY_LEVELS` block. it is implied through use of the Prior Object.

Prior distributions vs initial values vs fixed values

At present we do not support translation of the Prior Object to NONMEM prior specification for use with the $BAYES estimation algorithm. The only supported Bayesian estimation tool currently is WinBUGS. While NONMEM accepts a mix of prior distribution specification and initial values for estimation, WinBUGS requires prior distributions for all parameters, or fixing parameters to a given value. A fixed value can be thought of as a probability mass function (pmf) on a single value. A fixed value for a parameter represents a very strong prior on the value of that parameter. Bounds on parameters should be handled using an appropriate ProbOnto distribution e.g. Beta, Gamma, Uniform, Half-Normal, Truncated-Normal.

`PRIOR_PARAMETERS` Block

The [`PRIOR_PARAMETERS`] block holds constants which may be used in the `PRIOR_VARIABLE_DEFINITION` or `PRIOR_VARIABLE_DEFINITION` blocks. This allows the user to specify the general form of the prior distribution in the `PRIOR_VARIABLE_DEFINITION` block and then examine sensitivity to prior choice by altering the values in the [`PRIOR_PARAMETERS`].

The [`PRIOR_PARAMETERS`] block should contain variable assignment statements.

```
  <VARIABLE> = <value>
```

Note that In the [`PRIOR_PARAMETERS`] block, the variable is assigned a value, not a list with attributes. As mentioned above, if a model parameter is to be fixed then it takes the value assigned in the [`PRIOR_PARAMETERS`] block. An attribute “fix=true” is not required.

For example (/Priors/UseCase1_PRIOR):

```
PRIOR_PARAMETERS{
  # prior on "THETA"
  MU_POP_CL = 0.2
  MU_POP_V = 10
  MU_POP_KA = 0.3
  MU_POP_TLAG = 0.75
  VAR_POP_CL = 1
  VAR_POP_V = 1
  VAR_POP_KA = 1
  VAR_POP_TLAG = 0.1

  # prior on "OMEGA"
  MU_R_CL = 0.2
  MU_R_V = 0.2
  MU_R_V_CL = 0
  DF_OMEGA = 2
  MU_OMEGA_KA = 1
  MU_OMEGA_TLAG = 1

  # prior on "SIGMA"
  a_POP_RUV_ADD = 1.1
  b_POP_RUV_ADD = 3
  a_POP_RUV_PROP = 1.1
  b_POP_RUV_PROP = 3
} # end PRIOR_PARAMETERS
```

PRIOR_VARIABLE_DEFINITION

Parametric distributions as priors

In the `PRIOR_VARIABLE_DEFINITION` block we set up the prior distributions for the `STRUCTURAL` and `VARIABILITY` parameters of the Model Object. All model parameters must have a prior distribution specified, or a constant value set.

If model parameters are correlated or have a multivariate distribution then it is common (although not mandatory) to specify multivariate prior distributions. To do so, the user is likely to need to specify vectors of means and matrices for covariances or correlations. The syntax for specifying vectors and matrices is given in section 9.1.4.7.

The `PRIOR_VARIABLE_DEFINITION` block can contain assignment, transformation and random variable definitions using ProbOnto definitions for distributions.

For example (UseCase1_PRIOR):

```
PRIOR_VARIABLE_DEFINITION{
  # prior on "THETA"
  lMU_POP_CL = ln(MU_POP_CL)
  lPOP_CL ~ Normal(mean = lMU_POP_CL, var = VAR_POP_CL)
  POP_CL = exp(lPOP_V)

  lMU_POP_V = ln(MU_POP_V)
  lPOP_V ~ Normal(mean = lMU_POP_V, var = VAR_POP_V)
  POP_V = exp(lPOP_V)

  lMU_POP_KA = ln(MU_POP_KA)
  lPOP_KA ~ Normal(mean = lMU_POP_KA, var = VAR_POP_KA)
  POP_KA = exp(lPOP_KA)
  lMU_POP_TLAG = ln(MU_POP_TLAG)
  lPOP_TLAG ~ Normal(mean = lMU_POP_TLAG, var = VAR_POP_TLAG)
  POP_TLAG = exp(lPOP_TLAG)

  # priors on "OMEGA"
  R_mat = [[ MU_R_CL, MU_R_V_CL;
             MU_R_V_CL, MU_R_V ]]

  TAU_CL_V ~ Wishart2(inverseScaleMatrix = R_mat, degreesOfFreedom = DF_OMEGA)
  OMEGA_CL_V = inverse(TAU_CL_V)
  PPV_CL = sqrt(OMEGA_CL_V[1,1])
  PPV_V = sqrt(OMEGA_CL_V[2,2])
  PPV_V_CL = OMEGA_CL_V[1,2]

  TAU_KA ~ Gamma2(shape = 0.001, rate = 0.001)
  PPV_KA = sqrt(1/TAU_KA)

  # prior on "SIGMA"
  invRUV_ADD ~ Gamma2(shape = a_POP_RUV_ADD, rate = b_POP_RUV_ADD)
  invRUV_PROP ~ Gamma2(shape = a_POP_RUV_PROP, rate = b_POP_RUV_PROP)
  RUV_ADD = sqrt(1/invRUV_ADD)
  RUV_PROP = sqrt(1/invRUV_PROP)
  } # end PRIOR_VARIABLE_DEFINITION
```

Note the following parameters have prior distributions assigned: POP_CL, POP_V, POP_KA, POP_TLAG, PPV_CL, PPV_V, COV_CL_V, PPV_KA, RUV_ADD, RUV_PROP.

Note that priors on between subject variability are given on the precision scale (= 1 / variance). This is to facilitate use in BUGS. Here the distributions used are Wishart and Gamma on the precision parameters, but Inverse-Wishart and Inverse-Gamma may alternatively be used on the variance-covariance matrix and variance parameters.

Note also that priors for POP_CL, POP_V, POP_KA and POP_TLAG could also be defined using logNormal1 distributions (using the ProbOnto distribution) instead of transforming and back-transforming using Normal(. . .) distributions.

To specify a matrix, we use double square brackets, and specify elements row-wise, separated with a semi-colon. To specify elements of a matrix we use the R convention of square bracket specifying row and column entries. For example:

```
R_mat = [[ MU_R_CL, MU_R_V_CL;
           MU_R_V_CL, MU_R_V ]]
```

To specify the first row and column entry (corresponding to MU_R_CL): `R_mat[1,1]`.

Note that a Gamma distribution is used to define the prior on the between subject variability for PPV_KA. This is a legacy from the early days of fitting hierarchical models in BUGS where Gamma priors were conjugate and easier to sample from. They have been somewhat discredited as prior choices. Recent literature has favoured Half-Cauchy priors on variance parameters of hierarchical models as they are robust against smaller numbers of subjects.(Gelman 2006)

Non-parametric and empirical distributions as priors – inline data.

As an alternative to parametric distributions as priors, the user can specify non-parametric distributions (specifying bins of values and probabilities for each bin) or empirical distributions (specifying data forming the basis of empirical sampling). Univariate and multivariate sampling distributions have been defined in MDL for non-parametric and empirical sampling distributions.

In both cases the source for the non-parametric or empirical sampling can be specified inline via the [`PRIOR_PARAMETERS`] block, or by referencing an external data source in the `PRIOR_SOURCE` block.

Non-parametric distribution specification with inline data.

To specify a non-parametric distribution, MDL has a distributions called NonParametric and MultiNonParametric. These map to the ProbOnto RandomSample non-parametric distribution definition. To specify the non-parametric distributions, the user must supply bins and probabilities for sampling. To specify this inline we create a vector (for NonParametric) or matrix (for MultiNonParametric) of bins and a vector of probabilities. These are specified in the [`PRIOR_PARAMETERS`] block.

For example (Priors examples, Example3421dep)

```
PRIOR_PARAMETERS{

  ...

  # For Non-Parametric distribution

  bins_POP_K_V =
  matrix(vector = [2.006510,2.045465,2.084421,2.123377,2.162333,2.201288,2.240244,2.279200,2.318156,2.3
                   5.050013,5.050013,5.050013,5.050013,5.064166,5.064166,5.064166,5.064166,5.078318,5.0
                   ncol = 2, byRow is FALSE)

  p_POP_K_V = [0.033333,0.100000,0.100000,0.200000,0.100000,0.066667,0.166667,0.100000,0.066667,0.06666
} # end PRIOR

PRIOR_VARIABLE_DEFINITION{
  # prior on "THETA"
  POP_K_V ~ MultiNonParametric(probability = p_POP_K_V, bins = bins_POP_K_V)
  POP_K = POP_K_V[1]
  POP_V = POP_K_V[2]
  ...
} # end PRIOR_VARIABLE_DEFINITION
```

Here a matrix of bins for POP_K and POP_V is created by specifying a vector of values and then defining the number of columns and method for filling the matrix. A vector of probabilities is also defined. Then in the `PRIOR_VARIABLE_DEFINITION` block the multivariate non-parametric sampling distribution is defined referencing the probabilities and bins. Finally the Priors for POP_K and POP_V are defined by referencing the elements of the POP_K_V vector.

Empirical distribution specification with inline data.

Similarly, the user can specify the data source for empirical sampling within the [`PRIOR_PARAMETERS`] block and then refer to this in defining the sampling distribution for the `PRIOR_VARIABLE_DEFINITION`.

For example (Priors examples, Example 3422)

```
PRIOR_PARAMETERS{
  data_POP_K_V =
    matrix(vector = [2.006510,2.045465,2.084421,2.123377,2.162333,2.201288,2.240244,2.279200,2.318156,2
                     5.050013,5.050013,5.050013,5.050013,5.064166,5.064166,5.064166,5.064166,5.078318,5
                     ncol = 2, byRow is FALSE)
  ...

} # end PRIOR

PRIOR_VARIABLE_DEFINITION{
  # prior on "THETA"
  POP_K_V ~ MultiEmpirical(data = data_POP_K_V)
  POP_K = POP_K_V[1]
  POP_V = POP_K_V[2]
  ...
} # end PRIOR_VARIABLE_DEFINITION
```

Again, the data source is defined by specifying a matrix of values for POP_K and POP_V and then using this as the basis for the MultiEmpirical sampling distribution. For the univariate Empirical sampling distribution, only a vector would be needed as the basis for sampling.

NON_CANONICAL_DISTRIBUTION

As an alternative to inline data specification for non-parametric or empirical sampling distributions, the user may reference and external dataset for bins and probabilities (for use with non-parametric sampling distributions) or data for the basis of the empirical sampling distribution.

PRIOR_SOURCE

Similarly to the `SOURCE` block within the Data Object, the `PRIOR_SOURCE` block is a named list providing the file name, format of the source data. However the `PRIOR_SOURCE` block adds an argument to the list to provide a vector of column names to be used in the data source for the sampling distributions.

The syntax is as follows:

```
PRIOR_SOURCE{
 <data source name> : { file = <"filename.csv">,
                        inputFormat is csv,
                        column = [<"variable name1", "variable name2",
                                  ... , "variable name k"]}
}
```

Multiple data sources may be defined within the `PRIOR_SOURCE` block.

INPUT_PRIOR_DATA

The PRIOR_SOURCE data objects can then be referenced in the `INPUT_PRIOR_DATA` block to define how the data file columns map to objects to be used in the `PRIOR_VARIABLE_DEFINITION` block sampling distributions. This is done using anonymous lists.

The syntax is as follows:

```
INPUT_PRIOR_DATA{
  :: { src = <PRIOR_SOURCE data variable>,
       vector | matrix = <**PRIOR_VARIABLE_DEFINITION object>,
       column = "<PRIOR_SOURCE data column name>" }
}
```

For example :

```
NON_CANONICAL_DISTRIBUTION{
  PRIOR_SOURCE{
    NonPar_K_V : { file = "Nonparametric_K_V.csv",
                   inputFormat is csv,
                   column = ["bins_k", "bins_v", "p_k_v"]}

    Emp_SIGMA : { file = "Empirical_Sigma.csv",
                  inputFormat is csv,
                  column = ["data_SIGMA2"]}
  }

  INPUT_PRIOR_DATA{
    :: { src = NonPar_K_V, vectorVar = p_k_v, column = "p_k_v"}
    :: { src = NonPar_K_V, matrixVar=bins_k_v, column = ["bin_k", "bins_v"] }
    :: { src = Emp_SIGMA, column = "data_SIGMA2", vectorVar = data_SIGMA2 }
    }
  }

PRIOR_VARIABLE_DEFINITION{
  p_k_v::vector
  bins_k_v::matrix
  data_SIGMA2::vector

  POP_k_v ~ MultiNonParametric(bins = bins_k_v, probability = p_k_v)
  POP_SIGMA2 ~ Empirical(data = data_SIGMA2)
  POP_K = POP_k_v[0]
  POP_V = POP_k_v[1]
  }
```

In the above example, two sources are specified – one giving non-parametric sampling bins and probabilities for K and V, the other providing values for SIGMA to be used in the empirical sampling distribution. In the definition of NonPar_K_V we want to read three columns from the `PRIOR_SOURCE` data file – "bins_k" and "bins_v" to specify the bins for K and V, and "p_k_v" to specify the probabilities for sampling these bins. In the definition of Emp_SIGMA we define the columns of the `PRIOR_SOURCE` data file to use as the basis of the empirical sampling distribution of SIGMA.

In the `INPUT_PRIOR_DATA` block we specify how the defined `PRIOR_SOURCE` information is to be mapped to vectors and matrices defined in the `PRIOR_VARIABLE_DEFINITION` block and used in the definition of the sampling distributions. Note that in the `PRIOR_VARIABLE_DEFINITION` block we must define the type of the objects p_k_v, bins_k_v and data_SIGMA2 (vector, matrix and vector respectively). POP_k_v is then sampled from a multivariate non-parametric sampling distribution with bins specified by the matrix bins_k_v and sampling probabilities by the vector p_k_v, while POP_SIGMA2 is sampled from an empirical sampling distribution with values held in the vector data_SIGMA2.

Chapter 7

Design Object

Design Object overview

The Design Object is intended to provide information for creating simulation or evaluation of trial designs, and design space information required for optimal design. For trial simulation or design evaluation, the trial design is fixed and should be completely defined. For optimal design the user should specify a design space to optimise over in the `DESIGN_SPACES` block.

The user should specify all required interventions (drug administrations), sampling (or observation) schedules, populations, covariate distributions, and design parameters. These are then combined in the `STUDY_DESIGN` block to completely define study design(s). For simulation and evaluation only one design should be specified. For optimal design, many study designs can be considered and the `DESIGN_SPACES` block will determine what attributes the optimisation algorithm will search over.

An elementary design is given by the combination of `INTERVENTION` + `SAMPLING`, and can describe the design for groups of subjects e.g. treatment arms, or can define a unique design for an individual. In optimal design the combination of treatment arms and numbers of subjects in each is called the population design.

INTERVENTION

The `INTERVENTION` block contains details of administration schedules and types and how these are combined to form treatment interventions. Note here that we distinguish between treatment interventions (including placebo treatment, non-medical treatments, drug-free periods) and treatment arms, since an arm of a study may be purely observational i.e. with no treatment intervention. For the optimisation task of optimal design, the `INTERVENTION` block defines the treatment definitions for the starting design (if required by the algorithm).

The syntax for this block is:

```
<name> : { type is bolus,
  input = <Model Object dosingTarget>,
  amount = <real value or vector of real values>,
  doseTime = <vector of real values>,
  ssInterval = <real value>,
  timeLastSSDose = <real value>,
  doseIntervalVar = <dosing interval variable reference>,
  lastDoseTimeVar = <time of last dose variable reference>
  }
```

In the above, `ssInterval` and `timeLastSSDose`, `doseIntervalVar`, `lastDoseTimeVar` are optional.

```
<name> : { type is infusion,
  input = <Model Object dosingTarget>,
  amount = <real value>,
  rate = <vector of real values >,
  doseTime = <vector of real values>,
  duration = < vector of real values >,
  ssInterval = <real value>,
  timeLastSSDose = <real value>,
  timeStopSSInfusion = <real value>,
  doseIntervalVar = <dosing interval variable reference>,
  lastDoseTimeVar = <time of last dose variable reference>
}
```

In the above, one of `duration` or `rate` *must* be present. If `doseTime` is not present, then `timeLastSSDose` or `timeStopSSInfusion` *must* be present.

```
<name> : { type is combi,
  combination = <vector of previously defined interventions>,
  start = <vector of start times>,
  end = <vector of end times>
  }
```

The `combi` type is used to combine existing (already defined) intervention schedules to form more complex dosing regimens.

```
<name> : {type is reset,
  reset = { variable = <Model Object dosingTarget variable>,
  resetTime = <real value>,
  value = <real value>}
}
```

The `reset` type is used to reset the dosing variable in the Model Object at a given time.

```
<name> : {type is resetAll}
```

The `resetAll` type is used to reset all dosing variables.

For example, defining a single oral administration to the GUT at time 0 for use with the warfarin example (/Design/UseCase1_design1_eval):

```
INTERVENTION{
  admin1 : {type is bolus, input = GUT, amount = 100, doseTime = [0] }
}
```

Another example showing single dose administration with an oral dose to the GUT and an infusion to the CENTRAL compartment (UseCase4_design1_eval):

```
INTERVENTION{
  admin1 : {type is infusion, input = CENTRAL, amount = 100,
          doseTime = 0, duration = 100 }

  admin2 : {type is bolus, input = GUT, amount = 150, doseTime = [0] }
}
```

A more complex example showing how interventions can be combined to define sequences of interventions and actions.

```
INTERVENTION{


  ivbolus : {type is bolus, input = CENTRAL, amount = 100, doseTime = 0 }
  oral4 : {type is bolus, input = GUT, amount = 150, doseTime = [0,24,48,72] }

  ivinf : {type is infusion, input = CENTRAL, amount = 100, doseTime = 0,
  duration = 1 }

  wash1 : {type is resetAll}

  ivoral : {type is combi, combination = [ivbolus, wash1, oral4],
  start = [0, 24, 48]}

  oralinf : {type is combi, combination = [oral4, wash1, ivinf],
  start = [0, 96, 120]}

}
```

The above example describes three different drug administrations – an IV bolus dose, four doses of oral drug separated by 24 hours, and an IV infusion over the space of 1 hour. It also describes a washout "event" which resets the amount in all dosing variables in the model. We can then combine these four interventions and actions into a treatment arm definition. So the ivoral arm receives the ivbolus dose at time zero, then a washout / reset at 24 hours followed by the four oral doses starting at 48 hours. Note that the doseTime argument in each intervention definition defines the dosing time relative to the respective value in the start argument of the combination definition. Similarly, the oralinf arm receives the four doses of oral drug, followed by a washout / reset event at 96 hours then the ivinf treatment starting at 120 hours.

That is:

Time	0	24	48	72	96	120
ivoral	ivbolus	wash1	oral4[1]	oral4[2]	oral4[3]	oral4[4]
oralinf	oral4[1]	oral4[2]	oral4[3]	oral4[4]	wash1	ivinf

Note that for vectors of times, the `seq(...)` function can be used to specify a sequence of times with starting time, stopping time and interval size (MDL_Reference Guide section 5.80).

SAMPLING

Similar to the `INTERVENTION` block, the `SAMPLING` block provides a means to describe observation schedules which can then be used for individuals, for treatment arms or in combination to define complex patterns of observation. For the optimisation task in optimal design, the `SAMPLING` block defines the starting design sample times (if required by the algorithm).

The syntax for this block is:

```
<name> : {type is simple,
  outcome = <Model Object OBSERVATION block variable>,
  sampleTime = <vector of real values>,
  numberTimes = <integer>,
  deltaTime = <real value>,
  blq = <real value>,
  ulq = <real value>,
  }
```

In the above definition, the `outcome` argument is mandatory, all other arguments are optional. `numberTimes` defaults to the length of the `sampleTime` vector. Either `numberTimes` or `sampleTime` should be used, not both. Note that numberTimes should only be used in the context of Optimal Design optimisation tasks and not for design evaluation or simulation. deltaTime is the minimum time between two samples (for use in optimal design).

To combine a number of sampling definitions, the user can use the following syntax:

```
<name> { type is combi,
combination = [<vector of previously defined sampling schedules>,
  numberTimes = <integer>,
  start = <vector of real values>,
  relative = < Boolean>
}
```

The combination argument is mandatory, other arguments are optional.

For an example of definition of “simple” sampling scheme (UseCase1_design1_eval):

```
SAMPLING{
  window1 : {type is simple, outcome = Y,
  sampleTime = [0.0001, 24, 36, 48, 72, 96, 120] }
  }
```

For an example of combining sampling schedules (UseCase3_design1_eval):

```
SAMPLING{
  winPK : {type is simple, outcome = CC,
  sampleTime = [0.0001, 24, 36, 48, 72, 96, 120] }

  winPD : { type is simple, outcome = PCA,
  sampleTime = [0.0001, 24, 36, 48, 72, 96, 120] }

  # implies concurrent start, both start 0 unless define times
  sampPKPD : {type is combi, combination = [winPK,winPD] }
}
```

In the above, the sampling schedules for winPK and winPD start concurrently (at time 0) – there is no vector of start times to offset winPK and winPD.

POPULATION

The POPULATION block initialises and defines population characteristics or distributions for use in defining covariate distributions within the Model Object COVARIATES block. The POPULATION block acts like a COVARIATES block for the Design Object, where the user defines covariate distributions and how these may differ for each study arm. The COVARIATES block in the Model Object defines how (observed) covariates are used in the model.

The syntax for the POPULATION block is:

```
<population_name> : { type is template,
  covariate = [<list of covariate definitions>] }
```

Each covariate definition should have the following syntax. For continuous covariates:

```
{cov = <covariate name>, rv ~ <ProbOnto distribution>}
```

For discrete covariates:

```
{catCov = <covariate name>, discreteRv ~ <ProbOnto distribution>}
```

It is possible to define an arm to contain only one level of a categorical covariate (defined in the POPULATION block) by using the following syntax:

```
{catCovValue = <categorical covariate category>}
```

Note that covariates defined in the POPULATION block should be declared in a DECLARED_VARIABLES block within the Design Object.

For example, for the Design Object to be used with UseCase5 which uses WT, SEX and AGE as covariates (UseCase5_design2_opt.mdl):

```
DECLARED_VARIABLES{
  ...
  SEX withCategories{female, male}
  }

POPULATION{
  default : { type is template,
    covariate = { catCov = SEX, discreteRv ~ Bernoulli1( probability = 0.5) }
      }

  arm2Pop : { type is template, covariate = { catCovValue = SEX.female } }
  }
```

In the above example, a default distribution is defined for the SEX covariate which has categories female and male and defines that 50% of subjects should be male. (The Bernoulli distribution defines probability of the second category). arm2Pop is defined as having ONLY female subjects.

Another example for the same model:

```
POPULATION{
  default : { type is template,
  covariate = [
    { catCov = SEX, discreteRv ~ Bernoulli1(probability = 0.5) },
    { cov = WT, rv ~ Normal1(
        mean =  piecewise{{ 70 when SEX == SEX.male; otherwise 60 }},
        stdev = 10)
      }
    ]
  }

arm2Pop : { type is template,
  covariate = [
    { catCovValue = SEX.female },
    { cov = WT, rv ~ Normal1(mean = 55, stdev = 5) }
      ]
    }

  }
```

In the above example, SEX is defined as having 50% probability of being male, and WT is defined as having a Normal1 distribution where the mean depends on the value of SEX. If SEX is male then the mean is 70, whereas if female then the mean is 60. Both populations have a standard deviation of 10 for WT. arm2Pop is exclusively female and WT is defined separately for this group.

The populations defined in the POPULATION block are used in definition of the study arms within the STUDY_DESIGN block. For example:

```
STUDY_DESIGN{
  arm1 : {
    armSize = 16,
    interventionSequence = [{ admin = admin1, start = 0 }],

    samplingSequence = [{ sample = window1, start = 0 }],

    population = default
    }

  arm2 : {
    armSize = 17,
    interventionSequence = [{ admin = admin1, start = 0 }],

    samplingSequence = [{ sample = window1, start = 0 }],

    population = arm2Pop
    }

  }
```

STUDY_DESIGN

The STUDY_DESIGN block defines how the SAMPLING, INTERVENTION and POPULATION blocks are used to define arms in a study design. For the optimisation task of optimal design, it defines the starting design (if required by the algorithm). For evaluation and simulation tasks it defines the study design to be evaluated or simulated.

The syntax for defining each arm is as follows:

```
<name> : {armSize = <integer value>,
  sameTimes = <Boolean>,
  occasionSequence = [{occasion=<vector>,
    level=<reference varLevel>,
    start=<vector>}],

  interventionSequence=[{admin = <INTERVENTION block list name>,
    start = <real value> }],

  samplingSequence = [{sample = <SAMPLING block list name>,
    start = <real value> }],

  population = <POPULATION block list name>

  totalSize = <integer value>

  numberSamples = <vector of integer>

  totalCost = <real value>

  numberArms = <vector of integer>

  sameTimes = <Boolean>
  }
```

Recall that the intervention (`doseTime` argument) and sampling times (`sampleTime` argument) specified are *relative*. Within the `STUDY_DESIGN` block definitions we can specify start times for these that define in study time, when each starts. Using this construct it is possible to define interventions and samples that happen concurrently, and also those that happen sequentially. If the start argument is not given in interventionSequence or samplingSequence then it is assumed to be zero and the start times are taken from the relevant `INTERVENTION` and `SAMPLING` block definitions.

`armSize` defines the size of each arm.

If multiple outcomes are defined, then sameTimes (if true) denotes that the same observation times are to be used for all outcomes.

Additionally and optionally, arguments may be given pertaining to the study design as a whole. These are defined using the set keyword and are comma separated:

`totalSize` defines the overall size of the study. This defaults to the sum of the individual armSize arguments.

`numberSamples` defines the constrain on the number of samples for one subject. If one value is given, the number of samples should be equal to that value for all designs. If several values are given then this defines the set of number of samples allowable in designs (primarily used in optimal design).

`totalCost` defines the total cost for the entire population design.

`numberArms` defines the constraint on the number of arms in the study. If a single value is given then the final design will have exactly the required number of arms. If several values are given then this defines the set of number of arms allowable in designs (primarily used in optimal design).

`sameTimes` defines whether the sample sampling times are to be used for all observed outcomes.

As an example of the `STUDY_DESIGN` block (UseCase5_design2_opt.mdl):

```
STUDY_DESIGN{
  arm1 : {
    armSize = 16,
    interventionSequence = [{ admin = admin1, start = 0 }],
    samplingSequence = [{ sample = window1, start = 0 }],
    population = default
    }

  arm2 : {
    armSize = 17,
    interventionSequence = [{ admin = admin1, start = 0 }],
    samplingSequence = [{ sample = window1, start = 0 }],
    population = arm2Pop
    }
  }
```

Another example showing how arguments like totalSize and numberSamples should be used (when optimising a study design):

```
STUDY_DESIGN{
  set totalSize = 40,
  numberSamples = 4

  arm1 : {
    interventionSequence = [{ admin = admin1, start = 0 }],
    samplingSequence = [{ sample = window1, start = 0 }],
    population = default
    }
  ...

  }
```

DESIGN_SPACES

The `DESIGN_SPACES` block defines which design elements should be optimised in finding the optimal design. A design space defines the possible values of the design variables specified in `SAMPLING`, `INTERVENTION`, `POPULATION` blocks. It should not be used with design evaluation (`EVALUATE` task in the Task Properties object) or simulation. If no design space is defined for a variable e.g. amount or sampling schedule then it is assumed fixed for the optimisation process.

The syntax for `DESIGN_SPACES` definitions is as follows:

```
<name> : {objRef = <named object in INTERVENTION, SAMPLING or POPULATION>,
  element is <variable from named object above>,
  discrete = <vector of values, with type dependent on original argument >,
  range = <vector of values defining upper and lower range of values to be
    explored, with type dependent on original argument>
}
```

Valid choices for element depend on the object type (`SAMPLING`, `INTERVENTION`, `POPULATION`) that is being referenced: `bolusAmt`, `infAmt`, `duration`, `sampleTime`, `numberTimes`, `covariate`, `numberArms`, `armSize`, `parameter`, `doseTime`.

For example (UseCase1_design2_optFW.mdl):

```
warfarin_PK_ODE_design = designObj{
  ...
  SAMPLING{
    window1 : {type is simple, sampleTime = [0.0001, 36, 96, 120], outcome = Y }
    }

  DESIGN_SPACES{
    DS1 : { objRef = [window1], element is sampleTime,
      discrete = [0.0001,24, 36,48,72,96, 120] }

    DS2 : { objRef = [window1], element is numberTimes,
      discrete = [4,5] }
    }
  ...
  }
```

In the above example the `DESIGN_SPACES` defines how the optimisation algorithm will investigate the optimal design based on the window1 defined within the `SAMPLING` block, examining designs with 4 or 5 sample points chosen from the sample times defined in `DS1` i.e. [0.0001, 24, 36, 48, 72, 96, 120]. The sample times defined in `window1` of the `SAMPLING` block are used as the starting design for optimisation.

Another example:

```
warfarin_PK_SEX_design = designObj{
  ...

  INTERVENTION{
    admin1 : {type is bolus, input = INPUT_KA, amount = 100, doseTime = [0] }
    }
  SAMPLING{
    window1 : {type is simple,
      sampleTime = [0.0001, 24, 36, 48, 72, 96, 120],
      outcome = Y }
    }

  STUDY_DESIGN{
    arm1 : {
    armSize = 16,
    interventionSequence = [{ admin = admin1, start = 0 }],
    samplingSequence = [{ sample = window1, start = 0 }],
      }

    arm2 : {
    armSize = 17,
    interventionSequence = [{ admin = admin1, start = 0 }],
    samplingSequence = [{ sample = window1, start = 0 }],
      }
    }

  DESIGN_SPACES{
    DS1 : { objRef = [window1], element is sampleTime,
      discrete = [0.0001, 24, 36, 48,72,96, 120] }

    DS2 : { objRef = [window1], element is numberTimes,
      discrete = [4,5] }

    DS3 : { objRef = [admin1], element is bolusAmt,
      discrete = seq(100,300,100) }

    DS4 : { objRef = [arm1, arm2], element is armSize,
      range = [0,30] }
    }

  }
```

In the above example several design space elements are defined. `DS1` defines possible sampling times for `window1`, while `DS2` defines that designs of 4 or 5 samples are to be considered. `DS3` specifies that the amount to be dosed should vary between 100 and 300 with steps of 100. Finally `DS4` specifies how the number of subjects in each arm should be in the range [0,30] (inclusive) for both arms.

DESIGN_PARAMETERS

This block defines parameter values (constants) required for use in defining the design or to pass as constants to the model. Values are defined by assignment = or by equation =

.

Variables defined in `DESIGN_PARAMETERS` can be used in definition of covariate distributions defined in Design Object blocks.

Recall the example above where the distribution of covariates was defined in the `POPULATION` block:

```
POPULATION{
  default : { type is template,
    covariate = [
    { catCov = SEX, discreteRv ~ Bernoulli1(probability = 0.5) },
    { cov = WT, rv ~ Normal1(
      mean = piecewise{{ 70 when SEX == SEX.male; otherwise 60 }},
      stdev = 10)
      }
    ]
  }

  arm2Pop : { type is template,
    covariate = [
      { catCovValue = SEX.female },
      { cov = WT, rv ~ Normal1(mean = 55, stdev = 5) }
      ]
    }
  }
```

We can define the population means for weight in each population in the `DESIGN_PARAMETERS` block and pass these values into the `POPULATION` block definitions.

```
DESIGN_PARAMETERS{
  maleMeanWT = 70
  femaleMeanWT = 60
  femaleMeanWTArm2 = 55
  stdevWT = 10
  }

POPULATION{
  default : { type is template,
  covariate = [
  { catCov = SEX, discreteRv ~ Bernoulli1(probability = 0.5) },
  { cov = WT, rv ~ Normal1(
      mean = piecewise{{ maleMeanWT when SEX == SEX.male;
        otherwise femaleMeanWT }},
      stdev = stdevWT)
        }
      ]
    }

  arm2Pop : { type is template,
    covariate = [
      { catCovValue = SEX.female },
      { cov = WT, rv ~ Normal1(mean = femaleMeanWTArm2, stdev = stdevWT) }
      ]
    }
  }
```

Chapter 8

MOG Object

The MOG Object is where the user defines the MDL objects required for a particular task: estimation, simulation, design evaluation or optimisation.

INFO

The `INFO` block provides a name and/or problem statement to the associated MOG. The name attribute populates the Name tag in PharmML, while the problemStmt attribute populates the Description tag in PharmML. This information can then be passed forward to target software that support names or problem statement definition. For example, NONMEM conversion uses the problemStmt attribute to populate the `$PROB` statement, while the name attribute is converted to metadata in the comment header of the control stream file.

By default the Name tag in PharmML is "Generated from MDL. MOG ID: ".

The `problemStmt` attribute can be set via the ddmore R package function `writeMDL( ... , problemStmt = "Problem statement text")`.

The syntax for the `INFO` block is:

```
INFO{
  set problemStmt = <text string> ,
  name = <text string>
  }
```

The statements can be comma separated or the set command can be used for each line.

For example (UseCase1_1.mdl):

```
INFO{
  set problemStmt = "my Problem Statement"
  set name = "10May2016 Task Properties check"
  }
```

OBJECTS

The `OBJECTS` block defines the objects (defined in the current .mdl file) that are to be used in defining the Modelling Objects Group for use in the desired task. The MDL-IDE checks that these named objects exist in the current file.

The MDL-IDE also uses the MOG Object to "tie together" variable definitions across objects – it checks that variables used in the model are defined. So for example, if the model expects a covariate called logtWT but this is not defined in the Data Object then an error is given. Without a MOG Object, no validation check of this type is possible. Without the MOG Object, the MDL-IDE can only perform rudimentary syntax checking of MDL statements. With the MOG Object defined the MDL-IDE can check that the resulting model will result in valid PharmML.

The syntax for statements in this block is :

```
<Object name within the current MDL file> : {type is <dataObj |
designObj | mdlObj | parObj | priorObj | taskObj>}
```

Note that in the MOG Object, the user must specify dataObj *OR* designObj; parObj **OR** priorObj. As stated previously, for simulation, design evaluation or optimisation the Design Object takes the place of the Data Object. Similarly for estimation with BUGS or other Bayesian estimation software the Prior Object takes the place of the Parameter Object.

For example (UseCase1.mdl):

```
warfarin_PK_ODE_mog = mogObj {
  OBJECTS{
    warfarin_PK_ODE_dat : { type is dataObj }
    warfarin_PK_ODE_mdl : { type is mdlObj }
    warfarin_PK_ODE_par : { type is parObj }
    warfarin_PK_ODE_task : { type is taskObj }
    }
 }
```

Mapping of variable names between MDL Objects

The current version of MDL requires that variable names in each object are consistently named. Future versions of MDL may allow mapping between variable names across objects.

Chapter 9

MDL Language Reference

The aim of this section is to provide the technical aspects behind the MDL language: documenting its syntax and semantics in detail. New users may wish to read the sections on Data Object, Parameter Object, Model Object, Task Properties Object and Model Object Group, and explore the MDL implemented in the Use Case examples first in order to familiarise themselves with how MDL is used to define models. The information in this section may be of more interest to users who are writing their own models using MDL and wish to know the detail of syntax and grammar implemented in the MDL-IDE.

Like many computational languages MDL has two layers. First is the syntactic layer, or core, that defines how the words and symbols of the language are combined together in meaningful ways. This is like the building blocks of the language, it's vocabulary, punctuation and grammar. Building on this foundation then is the second, semantic layer of MDL. This is where the meaning of the language is defined and how the building blocks of the core are used to create a language that describes pharmacometric models.

This organisation is reflected in this section, which starts with the description of the language core, followed by an explanation of MDL's type system before moving on to the semantics layer of MDL.

Core syntactic elements

The core units of the language are described here from the bottom up. Starting with the language keywords, through the definitions of expressions and statements until we reach the highest level of organisation in MDL, the object.

Keywords

The keyword names are reserved and cannot be reused elsewhere, for example as attribute or variable names. The keywords in MDL have deliberately been kept to a minimum and at present there are 16. They are:

```
as, if, else, elseif, ln, exponentiale, false, in, inf, is, otherwise,
pi, piecewise, set, then, when, withCategories, true
```

include a keyword that are not currently used, but which is reserved for future versions of MDL:

```
ordered
```

Variable names

Variables names in MDL must conform to the following rules:

- There are no reserved variable names in MDL
- Variable names may only contain letters or numbers and ‘’ *and must start with a letter or* ‘’ character.
- As MDL is a case sensitive language the case of letters matters in variable names so ‘t’ is a different variable to ‘T’.

In technical terms a variable name must comply with the following regular expression:

```
('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..'Z'|'_'| '0'..'9')*
```

An addition constraint not reflected in the above regular expression is that the variable name also cannot begin with ‘MDL___’. This prefix is reserved for internal used by code generators that may be used to convert MDL to other languages and may need to create synthetic variable names.

Literals

Values such as numbers and strings etc can be written explicitly in MDL. Technically such values are referred to as literals. MDL supports the following types:

- Vector: e.g. [0, 1, 2, 25.0]
- Matrix: e.g. [[0, 20, 2; 23.5, 20, 1]]
- Strings: e.g. “a string“
- Integers: e.g. 99, 22, 0, -1, -477
- Real: e.g. 99.9, -0.473, 9e-2, -0.3424e5
- Boolean: true, false

Note that in the current MDL a vector containing only integer values will be regarded as a vector of type integer (not of type real). To ensure that the vector is of type real, the user may need to specify one of the numbers as a real value e.g. 25.0.

Expressions

An expression in MDL is primarily used to express mathematical concepts and evaluate mathematics. Expression are divided into two types, Boolean and numerical. The former is an expression that evaluates to a Boolean (True or False), while the latter evaluates to a Real number. Examples are:

```
x > 5 && y <= 0

x - 5 * 23

x^(2*x/z)
```

Expressions can contain mathematical functions too:

```
sin(x)

ln(x + y) + ln(22)
```

Some functions can use named arguments:

```
x + func(arg1=1, arg2=3)
```

In MDL conditional statements are also expressions:

```
x * if(y > 22) then 300 * a else 1
```

And they can use categorical variables:

```
x * if(sex == sex.female) then 1 else 0
```

Numerical and Boolean Expressions

Expressions are built up using operators that either take one or two operands:

binary_op :=

unary_op :=

Logical expressions are formed using combinations of Boolean operators (&&, ||, !) or comparison operators (<, >, <=, >=, !=, ==). Numerical expressions use the standard mathematical operators (+, -, /, *, ˆ, %). These are shown below.

Operator	Symbol	Left Type	Right Type	Result
Logical AND	&&	Boolean	Boolean	Boolean
Logical OR	||	Boolean	Boolean	Boolean
Less than	< R	eal R	eal B	oolean
Greater than	> R	eal R	eal B	oolean
Less than or equal	<= R	eal R	eal B	oolean
Greater than or equal	>= R	eal R	eal B	oolean
Equal	==	Real	Real	Boolean
Not equal	!=	Real	Real	Boolean
Power	ˆ R	eal R	eal R	eal
Multiplied by	* R	eal R	eal R	eal
Divided by	/	Real	Real	Real
Modulo (remainder)	%	Real	Real	Real
Add	+	Real	Real	Real
Subtract	-	Real	Real	Real
Negation (unary)	-		Real	Real
Positive (unary)			Real	Real

The operators have the same operator precedence you would expect in a standard mathematical equation. In the table below operator precedence is shown, ordered from highest to lowest.

Operators	Precedence
Unary	+ - !
Power	ˆ
Multiplicative	* / %
Additive	+ -
Relational	< > <= >=
Equality	== !=
Logical AND	&&
Logical OR	||

Variable references

A symbol name used in an expression is treated as a reference to a symbol defined elsewhere in the same object.

```
a = 10
```

```
b = 10 + a
```

The above code snippet shows how the variable reference ‘a’ in the expression refers to the definition of variable “a” which is initialised to 10. This is intuitive as is the fact the expression evaluates to 20. However, references to categorical variables behave slightly differently. A reference to it takes two forms:

1. A reference to the variable itself, e.g. sex
2. Or a reference to a category value, e.g. sex.female

Note that the second form uses a qualified name based on a combination of the categorical variable and its value:

.

The meaning of a reference to a category variable is self-evident, however, a reference to the categorical variable itself is less so. Consider the following:

```
sex withCategories {male, female}
sex == sex.female
```

In essence, the reference to the categorical variable ‘sex’ is referring to the category value held by ‘sex’. If ‘sex’ holds the value ‘sex.female’ then this expression evaluates to true. This enables us to write conditions expressions (see below) like this:

```
if(sex == sex.female) then 1 else 0
```

Type Specifications

As is described below MDL (see Type System) infers the type of parameter or variable where it can. However, when a variable is not initialised the author must tell MDL what type to expect. The type specification syntax enables this. Simple examples are below:

```
A::real

B::string

C:: int

D::pdf

E::Boolean
```

The basic syntax is ‘::’ followed by the type name. Vectors, and matrices require more complex handling:

```
F::vector[::int] # vector of ints

G::matrix[[::string]] # matrix of strings

H::matrix[[ ::vector[boolean] ]] # matrix of vectors of booleans
```

As you can see vector types need a type specification to define its element type as does the matrix. In order to reduce typing some type specifications can be omitted in which case the type is assumed to be Real:

```
A # type real

F::vector # vector of reals

G::matrix # matrix of real
```

```
H::matrix[[ ::vector ]] # matrix of vectors of reals
```

Conditional expressions

Sometimes it is useful for an expression to evaluate to different values depending on some arbitrary criteria. In a mathematical expression this is handled by a piecewise function:

$$f\left(x\right)=\left\{\begin{array}{ll} -1, & \text{x} < 0 \\ 1, & \text{otherwise} \end{array}\right.$$

In MDL we have a directly equivalent peicewise construct:

```
piecewise {{ -1 when x < 0; otherwise 1 }}
```

This can take more than 1 condition as you would expect:

```
piecewise {{ -1 when x < 0; 1 when x > 0; otherwise 0 }}
```

$$f\left(x\right)=\left\{\begin{array}{ll} -1, & \text{x} < 0 \\ 1, & \text{x} > 0 \\ 0, & \text{otherwise} \end{array}\right.$$

Or just condition and no otherwise clase:

Expressions with more than one condition are possible:

```
piecewise {{ -1 when x < 0; 1 when x >= 0 }}
```

which is equivalent to:

$$f\left(x\right)=\left\{\begin{array}{ll} -1, & \text{x} < 0 \\ 1, & x \geq 0 \end{array}\right. \left(\#eq:piecewise-Equation1\right) \tag{9.1}$$

$$f\left(k\right)=\binom{n}{k}p^{k}\left(1-p\right)^{n-k}\left(\#eq:binom\right) \tag{9.2}$$

A conditional expression should cover all possible conditions in order to prevent the generation of an undefined value, which will result in a runtime error. MDL does not enforce this, but to help ensure this it requires the writer provide at least two clauses. Ideally the last clause will be an 'otherwise' as this guarantees all conditions are covered, but in cases such as @ref(eq.piecewiseEquation1) this clearly is not necessary so this is not mandatory.

A related rule is that the conditions should also not define overlapping domains. In other-words only one condition can be true for any given set of values. This is illustrated by the code snippet below which breaks this rule:

```
piecewise {{ -1 when x < 0; 1 when x < 2; otherwise 0 }} # bad
conditions!
```

The first two conditions can be true if x is -1 for example, which makes the correct evaluation of this expression impossible (remember that the written order of the conditions is not meaningful). This last rule is very important, because the order that the conditions are evaluated cannot be guaranteed and so the result of the above expression may not be as expected. At the moment MDL does not check this so the author must ensure that all conditions are independent.

Complementary to the piecewise expression is the if/else expression. Note that this is an *expression* not a statement so it is evaluated as part of a mathematical expression and is not used to control which statements are evaluated, as it would be in an imperative language such as R. The syntax is as you would expect:

```
if(x < 0) then -1 elseif(x > 0) then 1 else 0
```

Here the order of evaluation is significant. So ‘x < 0’ is evaluated before ‘x > 0’. This means that the above example in an if/else expression is unambiguous:

```
if(x < 0) then -1 elseif(x < 2) then 1 else 0
```

as the condition ‘x < 2’ will never be evaluated. If/else expressions can be nested too:

```
if(x < 0) then -1 else if(x > 0) then 1 else 0
```

which is functionally equivalent to the ‘elseif’ non-nested equivalent.

Conditional Lists

A varant of the condition expression is the conditional list. This allows the conditional assignment to a variable of a list. For example:

```
A : if(B > A) then { type is linear, pop=POP_A, ranEff = ETA_A }
 else { type is general, grp=GRP_A, ranEff = ETA_A }
```

This assign one or other of the lists to A depending on the condition. This follows the same rules as for conditional expression, but the list being assigned must have the same List Super-type (see Type System, below). In the above example both lists have a super-type of ‘IndivAbstractList’ so the list assignment ‘:’ is valid.

A piecewise variant is also permitted:

```
A : piecewise {{
{ type is linear, pop=POP_A, ranEff = ETA_A } when B > A;
 otherwise { type is general, grp=GRP_A, ranEff = ETA_A }
 }}
```

Functions

Functions take two forms in MDL. Simple and named argument functions. The latter is equivalent to a standard mathematical function such as sin(a). The argument order is significant and all arguments are required. So for example:

```
logx(y, b)
```

defines a logarithm of y to base b. Swapping the arguments around would change the meaning accordingly.

The second form as one might expect takes named arguments and consequently the order of arguments is not important and some arguments are optional. For example:

```
foo(arg1 = val1, arg2 = val2, arg3 = val3)

foo(arg1 = val1, arg2 = val2)
```

call the same function, but ‘arg3’ can be omitted. Typically the function will use a default value, but the exact behaviour is determined by the definition of the function. Arguments can be constrained so that only specific combinations are permitted as below:

```
foo(arg1 = val1, arg2 = val2, arg3 = val3)
foo(arg1 = val1, arg4 = val4, arg5 = val5)

foo(arg1 = val1, arg4 = val4, arg2 = val2) # invalid

foo(arg1 = val1, arg5 = val5) # invalid
```

Here the combinations of arg2, arg3 and arg4 and arg5 are permitted, but combination such as arg2 and arg4 are not. Likewise, arg5 cannot be used unless arg4 is also provided. This gives a lot of flexibility in the parameterisation of functions.

Function references

When a function is invoked a reference to that function's definition is invoked and the appropriate value returned. For example, in the expression:

```
 a = ln(10)
```

the symbol 'ln' is a reference to the function defining the natural log.

However, in some circumstances it is desirable to refer to a function, but not to evaluate it immediately, for example when referring to a function to be used for interpolation of data. In these circumstances a '&' precedes the function name and the function arguments are omitted. For example:

```
Obs : { use is dv, variable = Y, interp=&linear }
```

Here the 'interp' attribute is assigned a reference to the 'linear' function definition. This function can then be invoked whenever interpolation between two data-points is required.

Vectors

Vectors are defined within in square brackets and can take numbers, strings, Boolean, variable references, logical and numerical expressions. Vectors must contain items of consistent type. The exception is when the vector contains numbers and variable references of numerical types (see type system below for more information). Some examples are below:

```
[ a, b-1, 1.0 * 3^5, 2.0^3.5, inf ]

[ "str", "a str too", "a", "str" ]

[ true, false, true == false ]
```

Note that all elements must be populated. So the following is *not* permitted:

```
[ 3, , 3 ] # forbidden
```

Vectors can also be nested, but again the vectors must have the same type:

```
[ [ 22, 45 ], [ 67, 89 ] ]
```

Vectors may be assigned to a variable or parameter and then used in other expressions:

```
A = [ 1.1, 2.2, 3.3, 4.4 ]

B = A # B is assigned the contents of variable A

B = A[1] # B is a scalar and assigned the first value in A: 1.1

B = A[2:3] # B is vector and assigned elements 2 and 3 in A: [2.2, 3.3]
```

This illustrates how vector indexing works. The first element is at position 1 and the last index is equal to the length of the vector. If a single index value is used in square brackets, then a single scalar value is returned. As can be seen, MDL permits ranges of values to be specified, using a ':' operator. This range operator is only permitted for vector and matrix indexing and ensures that the indexed vector returns a vector. This has a useful side effect when a vector containing a single vector is required:

```
B = A[3:3]
```

this returns a vector containing just element 3 of vector A, i.e. [3.3].

Vector index ranges can be omitted too. This indicates that indexing should start at the beginning or the end of the vector. For example:

```
A[:3] # first element to element 3

A[2:] # element 2 to the last element

A[:] # first to last elements, i.e., the complete vector
```

Note that the last index is equivalent to the vector reference:

```
B = A[:]

B = A # equivalent to above
```

Matrices

Matrices are very similar in their semantics to vectors and are indicated syntactically by the double square bracket with each row terminated by a ';':

```
M = [[ 1, 2, 3; 3, 4, 6 ]]
```

Like vectors all cells in a matrix literal must be populated and in addition all rows must have the same number of columns:

```
[[ a, b, c;
 d, e; # forbidden - inconsistent num columns
 g, h, i ]]

[[ a, b, c;
 d, , f; # forbidden - missing value
 g, h, i ]]
```

Matrix indexing is similar to vector indexing and uses the same square bracket operator. Instead, an index for the row and column are specified:

```
M = [[ 1, 2, 3; 4, 5, 6 ]]

N = M[2, 3] # a scalar value of 6
```

The first index indicates the row number and the second the column. As with vectors indexing starts at 1. Range operators behave in the same way as vectors, but in this case they always return a matrix:

```
N = M[2, 2:3] # returns matrix: [[5, 6]]

N = M[1:2, 2] # returns matrix: [[2; 5]]

N = M[1:2, 1:2] # returns matrix: [[1, 2; 4, 5]]

N = M[1:1, 2:2] # returns matrix: [[2]]
```

Empty ranges behave as with vectors, indicating the start or end of a row or column. Row or column indexes can also be empty to indicate all rows or columns:

```
N = M[, 1] # returns matrix: [[1; 4]]

N = M[, 1,2] # returns matrix: [[1, 2; 4, 5]]

N = M[2,] # returns matrix: [[4, 5, 6]]

N = M[,] # equivalent to N = M
```

Sublist Expressions

The sublist is a convenient way of grouping together related pieces of information together in an expression. In its basic form the sublist is a set of attributes combined together as below:

```
{ att1 = val1, att2 = val2, ... }
```

Sublists which have different attribute sets are regarded as different. In fact sublists are fully fledged types in MDL (see section) so sublists with different attributes sets are distinct types. This is illustrated in the example below where th sublists correspond to sublist types a and b:

```
 { att1 = val1, att3 = val3 } # sublist a

 { att2 = val2, att4 = val4 } # sublist b
```

This means that an argument in a function (see section for more detail on argument typing) can require a particular sublist type. In this way type system can ensure that only the valid type system is used. For example if an attribute 'foo' expects a sublist of type a, then the following validity is enforced:

```
 foo = { att1 = val1, att3 = val3 } # sublist a - valid

 foo = { att2 = val2, att4 = val4 } # sublist b - invalid
```

How is the type of a sublist determined? Very simply all sublists must have a unique set of attributes. The MDL processor takes the combination of attributes written and matches them to a sublist in its dictionary of sublist types.

Sublist can also restrict the set of permitted attributes, just like you can with named function arguments (see section 9.1.6.3). The details of how this is carried out is described in detail below (section 0). The sublist can be contained in a vector as well. For example:

```
 [

 { att1 = val1, att2 = val2},

 { att1 = val3, att2 = val4},

 ]
```

A Sublist can be used by any attribute, function argument or property. Below is an example of its used in a function. It shows how the sublist provides a convenient way to group together sets of covariate and fixed effect parameters in a linear individual parameter definition:

```
 ln(CL) = linear( trans is ln, pop = POP_CL, fixEff = [
  {coeff = BETA_CL_WT, cov = logtWT},
  {coeff = POP_FCL_FEM, catCov = SEX.female },
  {coeff = BETA_CL_AGE, cov = tAGE}
 ],
 ranEff = [ETA_CL] )
```

Variable selection expression

MDL provides support for conditionally assigning values in a variable to another variable using a special syntactic structure called a *variable selection expression.* This construct is often used in a list (described in section) and is best illustrated by an example:

```
CMT: { use is cmt }

AMT: { use is amt,
 define={1 in CMT as GUT, 2 in CMT as CENTRAL} }
```

The value mapping syntax is used with the 'define' attribute. It can be read as "if value is '1' **in** variable 'CMT' then select **as** variable 'GUT', if value is 2 **in** 'CMT' then select **as** variable 'CENTRAL' ". The semantics of what selection means can vary depending on context. In the above example, which is valid MDL, the lists AMT and CMT each define a column in a dataset with the AMT column values being assigned to the variable selected by the corresponding value in the CMT column.

The syntax for the variable selection expression is as follows:

```
 { <test value> in <qry var ref> as <select var ref>,
  <test value> in <qry var ref> as <select var ref>, ... }
```

The query (qry) and selection (select) variable references cannot be the same, the same query variable must be used throughout the expression and the test value must be a numerical value.

Category value selection expression

Related to the variable selection expression is the category value selection. This carries out a similar function, but for category values, in this case a particular category value is selected when a given expression is matched. For example:

```
 { sex.female when 0, sex.male when 1 }
```

shows how sex.female is selected when the value is 0, and sex.male when it is 1. Typically this expression is used in conjunction with a variable selection expression where the value in a list is to be mapped to a categorical variable. This is illustrated below where the values in a DV column definition are mapped to either a variable or a set of category values:

```
 DVID: { use is dvid }

 DV: { use is dv,
  define={1 in CMT as GUT,
  2 in CMT as { Outcome.dead when 0, Outcome.alive when 1}
  }
 }
```

The basic syntax is:

```
 { <category.value> when <test value>,
  <category.value> when <test value>, ... }
```

The test value should be a numerical value and the category values belong to the same category.

Attributes, Arguments, Properties and Values

Attributes in lists, arguments in functions and properties all behave in the same way in MDL. For the sake of simplicity this description will refer to attributes, but this should be understood as a synonym for argument.

An attribute is simply an identifier that is associated with a value. That value can be of any valid type and is usually assigned to the with the '=' operator. For example:

```
att1 = 0

att2 = true

att3 = "string val"

att4 = { a = 0, b = 3}
 # sublist

att5 = [0, 2, 3]

att6 = [1.5, inf, x]

att7 = { 1 in CMT as foo }
 # a mapping

att8 = varRef
```

In addition an attribute can be assign a value from a controlled vocabulary of options, called a built-in enumeration. Because the MDL parser needs help to distinguish these option names from a variable reference it is necessary to use a different assignment operator. Therefore, we use the keyword 'is' to indicate that an attribute has been assigned an option. For example:

```
att8 is anOption
```

A typical usage of a built-in enumeration is to define the key value of a list, for example:

```
c1 : { use is id }

c2 : { use is adm, variable = D }

c3 : { use is idv }
```

Any name can be used as an attribute name as long as it is not a language keyword and it is a valid variable name.

In the previous version of MDL an attribute expecting a vector type would have to be written thus:

```
A : { foo = [ z ] } # foo expectes a vector
```

even when there was only one element in the vector. To avoid the additional typing and to improve readability it can now be written as:

```
A : { foo = z }
```

The 'foo' attribute still expects a vector, so behind the scenes MDL converts this value to [z].

Statements

The statement is the core of MDL and comes in several forms. However, they all have the following characteristics:

1. A statement can be split over any number of lines. The parser detects the start and end of the statement based on its context.
2. Sometimes it is helpful to the user to indicate where a statement starts and ends in which case an optional ';' character can be used. Note this is completely optional.

The different statement types are below.

Equation definition

This defines a variable using a notation equivalent to a mathematical equation. It can have three forms:

1. x = `x = 2 + 5 / ln(22)`
2. fn(x) = , where x is transformed by a function `ln(x) = 2 + r`
3. x, where the variable x is declared but not initialised. `x`

The symbol (parameter or variable) it defines always has a type of Real.

In example 2 above, a transformation was used on the variable x. It is important to note that the variable x can be used later (without the transformation) and it is implied that a back-transformation will have been applied. The user need not explicitly back-transform the parameter in the MDL code.

Category definition

A category definition creates a variable that has two roles. First it groups together a set of categories that belong to this variable and second it holds a value that is one of these categories. Exactly what this means is explained below, but here is the syntax of the category definition.

```
X withCategories { cat1, cat2, ... catN }
```

The definition can create any number of categories but must have at least one. A simple example is:

```
sex withCategories { male, female }
```

The category values are specific for each category variable so the following is permitted:

```
sex1 withCategories { male, female }

sex2 withCategories { male, female }
```

The categorical variable always has a type of Enum.

List Definition

The list is a way of associating attributes and values with a variable. In many ways it is similar to a class seen in an Object Oriented programming language. The list has the following syntax:

```
lst : { keyAtt (=|is) <value>, att (=|is) <value>, ... }
```

Note that a list holds a specific set of attributes. The exact set is determined by the key attribute used, it's value and the block containing the list. This is illustrated below.

```
BLK1{
lst1 : { key is val1, att2 = val2, att3 = val3 }
lst2 : { key is val2, att20 = val20, att3 = val3 }
}
```

In the above example the attribute named 'key' is the key attribute in block BLK1. Note that a block can have only one key attribute. That means, as in this example, the value may be used to distinguish between lists. So when the key attribute has a value of val1 the list uses a different set of attributes compared to when the value is val2.

```
BLK2{
lst3 : { key = keyVal, att2 = val2, att3 = val3 }
lst4 : { key = keyVal, att2 = val2, att3 = val3 }
}
```

BLK2 by contrast is configured not to use the value of the key attribute so each list must use the same set of attributes. Note that the attribute names can be the same across lists and the same key attribute name can be used in different blocks. Attribute names cannot be repeated *within* a list however and the key attribute is always mandatory.

Anonymous lists

The anonymous list is a legitimate version of a list, but it does not define a named list variable. This is used where one wishes to group together a set of attributes that are related to each other, but when we do not want the list to be referred to elsewhere. Its rules and behaviour are identical to this of the list in every other respect. Its syntax is as follows:

```
:: { keyAtt (=|is) <value>, att (=|is) <value>, ... }`
```

Note the '::' symbol which designates this as an anonymous list.

Random variable definitions

Random variables are defined using the '~' assignment operator which is the common mathematical convention when relating a random variable to a probability distribution. In most respects the random variable definition behaves like a standard equation definition except that the expression on the right hand side of the '~' must have a type of Pdf. Note that the variable can have a transformation function on the left hand side. This is illustrated in the examples are below:

```
ETA_CL ~ Normal(mean = 0, sd = PPV_CL) # valid

ln(ETA_CL) ~ Normal(mean = 0, sd = PPV_CL) # valid

ETA_CL ~ PPV_CL # invalid
```

The generic syntax description is as follows:

```
<ID> ~ <PDF expression>
```

Category Lists

It is possible to define a list that also defines a set of a categories. This type of definition allows the writer to associate a category definition with other attributes and information. It also supports the ability to select a category based on an expression. This is illustrated in the example below:

```
SEX: { use is catCov withCategories { M when 0, F when 1 } }
```

where the list definition, SEX, can be treated as a categorical variable with categories 'M' and 'F'. This list also defines a data column so the 'when' syntax indicates that the 'M' category value is assign to SEX if the data value is 0 and the 'F' category value if 1. This provides a shorthand that allows us to both define the categories and their mapping. The syntax of a category list is:

```
<ID>: { <attName> is <builtinEnum>
withCategories { <cat value> when <selection expr>,
 <cat value> when <selection expr> } [,
<attName> [is|=] <cat value>, ...] }
```

Property Definitions

Sometimes it is desirable to define properties that we wish to associate with a whole object. Sometimes this is because we want to define default attribute values for statements with the block or to set properties that only need to be defined once and which do not need to be referred to in an expression. To support this MDL provides the property syntax as follows:

```
set <prop name> [=|is] <prop value> [,
 <prop name> [=|is] <prop value>, ...]
```

An example of this can be found in the task object where the properties of the task, such as the estimation algorithm are set in this way:

```
ESTIMATION{
 set algo is saem
}
```

The property name is specific to the block and is unique to the block, so repeating property definition is forbidden. So:

```
## INVALID!
 ESTIMATION{
  set algo is saem,
     algo is focei
 }
```

or

```
## INVALID!
 ESTIMATION{
  set algo is saem
  set algo is focei
 }
```

will result in an error. In all other respects property names behave just like attributes within a list, they can have the same types, they can be optional and mandatory and can be constraint to only permit certain combinations of property.

Blocks

The block is used to organise similar concepts and it can be configured to only contain certain types of statement. As we have seen above the block also provides the context for what list and property attributes are available for use. The generic syntax is

```
blkName [(name=value)] { <statement> [;] <statement> [;] ... }
```

In the above syntax description a statement may also be another block and so in this way sub blocks may be nested within each other. In MDL such nesting is limited to one level as can be seen in the example below:

```
MODEL_PREDICTION {
 DEQ{
   RATEIN = if(T >= TLAG) then GUT * KA else 0
   GUT : { deriv =(- RATEIN), init = 0, x0 = 0 }
```

```
    CENTRAL : { deriv =(RATEIN - CL * CENTRAL / V) }
    }
  CC = CENTRAL / V
 }
```

where the DEQ block is used to contain the definition of differential equations.

Blocks constraint the statements they contain in the number of ways:

1. by type. Some blocks may only permit lists or equation definitions or combinations of statement types.
2. by count. Each block defines the minimum and maximum number of statements it can contain.
3. by sub-block. The block may permit no sub-blocks or sub-blocks with specific names.

Objects

The object is the highest level of syntactic organisation in MDL. It defines a container for a set of blocks and the variables defined inside them. In MDL the object has a specific purpose and its semantic are a combination of that purpose and the semantics of the blocks and statements it contains. Its generic syntax is below:

```
 <ID> = <objName> { <block> [, <block>, ... ] }
```

where the objName is an internal MDL identifier for the type of the object. The object's type is related to semantic purpose. Note that the object type controls what types of block it contains. For example in MDL an obj of type 'dataObj' cannot contain an 'IDV' block. A short example of a dataObj is given below:

```
warfarin_PK_ODE_dat = dataObj {

  DECLARED_VARIABLES{GUT Y}

  DATA_INPUT_VARIABLES {
    ID : { use is id }
    TIME : { use is idv }
    WT : { use is covariate }
    AMT : { use is amt, variable = GUT }
    DV : { use is dv, variable = Y }
    } # end DATA_INPUT_VARIABLES

 SOURCE {
   srcfile : {file = "warfarin_conc.csv",
   inputFormat is nonmemFormat }
   }

 }
```

The Type System

One of the core mechanisms for ensuring correctness in MDL is its type system. In this section we explain what the different types are, and any rules associated with their correct usage.

The types

Name	
Int	
Real	
Boolean	
Enum	An enumeration type. A namespace for categori
Enum value	A specific enumeration defined by an en
Builtin Enum	Commonly abbreviated to BE, this is a predefined set of enumerated values that are part of the MDL d
List	A data structure that associa
List Super-Type	An abst
Pdf	A Probability density functi
Random Variable	A value that is a random variable and was obtained from a Probability Distribution. It ca
String	
Pmf	A probability mass function type.
Mapping	This is the t
Sublist	This is a sub
Vector	A one dim
Matrix	
Reference	A reference
Undefined	No type. This is used internally to indi

The default type

In MDL the default type is Real. In a standard equation or random variable statement the symbol defined on the LHS of the definition is always of Real type. Examples:

```
A = <expression>

ln(B) = <expression>

C ~ <expression>

D
```

Type promotion

MDL allows an integer type to be used in mathematical expression. It does this using type promotion, where the integer value is automatically converted to a real value. This gives the kind of behaviour that the reader would expect. For example:

```
A = 22.55 + 1

A = 22.55 + 1.0
```

are equivalent. Note that mathematical expressions always evaluate to a value with a Real type so:

```
A = 2 * 55
```

is effectively evaluated as:

```
A = 2.0 * 55.0
```

Typing of more complex types

Vector type

A vector can potentially have elements of any type and in general all its elements must be of the same type. We refer to a vector type as "vector of type X" or an "X vector". For example, vector of type String or String vector.

The type promotion rules above also apply to vectors, which means that a Real vector can contain a mixture of integer and real values.

Note that when writing a vector literal (see above) the type is inferred from its content. This means that for a vector to be of type Real it must contain at least one Real value as can be seen here:

```
[ 0, 2, 3, 4 ] # Int vector

[ 0, 2.0, 3, 4 ] # Real vector

[ -1.0, 2.0, 3.0, 4.0 ] # Real vector

[ true, false ] # Boolean vector

[ "A", "b", "C" ] # String vector

[ { att1 = val1 }, { att1 = val2 } ] # Sublist vector
```

Matrix type

This behaves identically to the vector type. The type pf the matrix is inferred from its contents in exactly the same way.

List type

The type of the list is defined by a combination of its

1. owning block
2. key attribute
3. key value

This is best illustrated in the example below. Here the 'c1' variable is of type 'List:Idv' and 'c2' of type 'List:Amt'. They both belong to the same block, the key attribute is 'use' so the discriminating factor in determining their type is the values 'idv' and 'amt'.

```
DATA_INPUT_VARIABLES {
  c1 : { use is idv , variable= GUT }
  c2 : { use is amt , variable= GUT }
  }

DATA_DERIVED_VARIABLES{
  c3 : { use is doseTime, idvColumn=c1, amtColumn=c2 }
  }
```

In the DATA_DERIVED_VARIABLES block 'c3' is of a different type again, but contains two attributes, idvColumn' and 'amtColumn' that expect references to variables of type 'List:Idv' and 'List:Amt' respectively. List types are very specific for referencing another column type (as below) would result in a typing error:

```
 DATA_DERIVED_VARIABLES{
# INVALID!
 c3 : { use is doseTime, idvColumn=c2, amtColumn=c1 }
 # type error
 }
```

In some cases the list value is not required to define the type. In the example below the type 'List:deriv' is specified by just the block, 'MODEL_PREDICTION', and the key attribute 'deriv'. As a consequence 'GUT' and 'CENTRAL' both have the same type.

```
MODEL_PREDICTION {
  DEQ{
    RATEIN = if(T >= TLAG) then GUT * KA else 0
    GUT : { deriv =(- RATEIN) }
    CENTRAL : { deriv =(RATEIN - CL * CENTRAL / V) }
    }
  CC = CENTRAL / V
  }
```

Each List type can potentially be converted to one another type when necessary. This is particularly useful if the semantics of the list make it desirable to use the list variables in an mathematical expression. The above example shows just such a case. The semantic of List:deriv lists is to define a difference equation, with the list variable corresponding to the derivative variable. The List:deriv type has a conversion type of Real. This means that when used in contexts that expect a Real type the Type system uses the conversion type. This allows the example above to be valid despite the fact that list variables are used as real values. All list type can potentially have **one** conversion type, but it's use is optional and if not define then the list cannot be converted.

In the latest version of MDL there is an alternative method of identifying a list. This is via the attribute name. This acts as the list key and must be unique across all list types associated with the same block. For example, we can differentiate between different parameters as follows:

```
A : { value = 0.2 }

A : { vectorValue = [0.2, 0.3] }

A : { matrixValue = [[ 0.2, 0.4; 0.5, 0.9 ]] }
```

The above lists each have different types identified by the attribute name used.

List super-types

Sometimes it is useful to group related lists and refer to them interchangeably. For example, observation variables use different list types to define different continuous error models:

```
Y1 : { type is combinedError1 ... } # list type combinedError1List

Y2 : { type is combinedError2 ... } # list type combinedError2List
```

However, they are clearly related and should be compatible with each other. This is where the list super type is useful. In the above case both list types share the same super type, 'observation'. This enables another attribute to be assigned this super-type and it will be able to refer to either of these variables:

```
Z1 : { superTypeAtt = Y1 } # valid

Z2: { superTypeAtt = Y2 } # valid
```

which would not be possible with normal list typing:

```
X1 : { listTypeAtt = Y1 } # valid as expects type is combinedError1List
```

```
X2: { listTypeAtt = Y2 } # invalid as expects type combinedError1List
```

In the same way super-types enable conditional list assignments that involve different list types:

```
U : if (x > 0) then { type is combinedError1, ... }
 else { type is combinedError2, ... }
```

here ‘U’ is allocated the type ‘observation’ because both lists share the same super-type. In this way MDL knows that these lists are compatible.

Built-in enumeration type

A built-in enumeration is essentially a controlled vocabulary that constrains the values that can be assigned to an attribute. Each built-in enumeration is different and consists of a set of strings. The enumeration is match if the name assigned to the attribute is one of the names held by its built-in type. As an example if there is a built-in type ‘eg’ that permits the names ‘foo’ and ‘bar’ then the following cases are valid and invalid:

```
 att1 is foo # valid

 att1 is bar # valid

# INVALID!
 att1 is ugg
# INVALID!
 att1 = foo
```

The last case is important to note. MDL only knows to expect a building enumeration if it is preceded by the ‘is’ keyword. In the last statement above the assignment symbol was used and in this case MDL would treat this as a reference to a variable.

Sublist type

Sublists are types and they are distinguished from each other by their attributes. This is best illustrated by an example:

```
 ln(CL) = linear( trans is ln, pop = POP_CL,
                 fixEff = [
                    {coeff = BETA_CL_WT, cov = logtWT},
                    {coeff = POP_FCL_FEM, catCov = SEX.female },
                    {coeff = BETA_CL_AGE, cov = tAGE}
                    ],
                 ranEff = ETA_CL )
```

Above the attribute ‘fixEff’ expects a vector type of ‘Sublist:FixEff’. The type system looks at the available sublist types and identifies the one that has the same attribute names. Note that the same type can allow different attribute combinations as can be seen in the example. If no subtype can be identified then the sublist is given a type of Undefined which will result in a typing error. Identifying the correct subtype also then allows the attributes in the sublist itself to be type checked.

Enum and Enum Value types

Enumeration types are unusual in MDL in that the type is to some extent defined within MDL. Take the following definitions:

```
 Gender1 withCategories { male, female, other }

 Gender2 withCategories { male, female, other }

 State withCategories {alive, dead }
```

The first defines an enumeration type with individual values, 'male', 'female' and 'other'. The second definition is distinct from the first because it is associated with a different symbol. So in a sense the above definitions have created 3 new types called Gender1, Gender2 and State. However, the above statements also define 3 variables of the same name and as variables they can be initialised with one of their enumeration values. So the variable 'State' can hold a value of 'alive' or 'dead', or more correctly 'State.alive' or 'State.dead'. This means that when used as a variable reference these variables always have a type of Enum value. This is illustrated by the example expressions below:

```
 Gender1 == Gender1.male # valid

# INVALID!
 Gender2 == Gender1.male

# INVALID!
 Gender1 == Gender2
```

Type inference rules for Symbols

When a variable is defined is must be allocated a type and unlike language such as R, this type cannot change. Where is can MDL works out the type of the variable, usually from the type of any expression on the right hand side of an assignment operartor ('=', '~' or ':'). Where there is no assignment the variable type defaults to Real unless the author explciltly defines its type using a type specification (see above). This is best describe using examples:

```
A = [ 1, 2, 3 ] # vector of ints

B # real

C::string # string

D = true # Boolean

E ~ Poisson1(rate=lambda) # Random Variable of int

F withCategories {heads, tails} ~ Bernoulli1(probability=p1)
 # Random Variable of enum

G = [[ "A", "B"; "C", "D"]] # matrix of strings
```

Scoping and statement ordering

MDL is declarative. It describes what the model is not how to implement it. In common with other declarative languages MDL has the following features:

- The order of blocks and statements within blocks is ***not*** significant.
- This means that symbols can be defined after they are referenced.
- A variable ***must not*** be assigned to more than once.
- In general a variable cannot be assigned to an expression that is dependent on itself. For example the following is not permitted:

```
a = b

b = c

c = a (this makes a cycle back to the first statement)
```

A list can be defined that is exempt from this rule. For example, a derivative list can refer to iself:

```
x : { deriv = -x }
```

This is consistent with mathematical definitions such as: dx/dt = -x

The scoping unit for MDL is the object. Variables defined inside an object are visible to expressions in the same object, but not outside it. This means that each MDL object is a self contained definition that does not rely on any other object.

MDL Reference Guide

The [MDL reference guide is available online] (https://github.com/ModelDefinitionLanguage/MDLUserGuide/blob/master/mdl_reference.pdf). This document gives detail of Object, Block, List, Sublist, Function and Type definitions. As it is based on and generated from the language definition file it documents exactly how the language is constructed and is definitive in describing what is valid MDL as implemented in the MDL-IDE. Elements of the Reference Guide have been extracted here in a format intended to help user understanding which may be used for quick reference.

How to read the MDL Reference Guide.

There is a hierarchy in the MDL Reference Guide. This reflects the way that users approach the definition of a model in a sequential fashion. Understanding this hierarchy will help the user identify what statements and attributes are valid as shown here:.

- Object definitions describe which Blocks are valid within each Object.
- Block definitions describe
  - what arguments are associated with the block
  - what types of statements can be made within the block
  - what sub-blocks are permitted within the block
  - what list types are to be used with the block
  - what properties are permitted within the block.
- List types define
  - whether the list can be anonymous i.e. using `::{ type is ... }`
  - whether the list can define categories
  - what types are associated with the list i.e. `type is ...`

  - what attributes are permitted withing the list.
  - what the "signature" of the list looks like, including identifying which attributes are optional.
  - NB: list type names do not necessarily match the names used in MDL e.g. in defining individual parameters we type `CL : {type is linear, ... }` but the type of the list in the Reference Guide is "IndivParamLinear". However the list type is given in the Block definition List table (left hand column) with the associated MDL key value in the right hand column.

- Sublist definitions define the (small number of) cases where list types contain other lists, for example in definition of the fixed effect model within a IndivParamLinear list. The sublist definition defines:
  - what attributes are permitted within the sublist
  - the permitted signature of the sublist and which attributes are optional.
- Function definitions define:
  - Arguments of the function
  - Type returned from the function.
- Type definitions define
  - the types and associated type classes.
  - the keyword types and associated enumeration types. e.g. `type is combinedError1`; `set algo is saem`.

Objects

Valid types of Objects are:

`dataObj, designObj, parObj, priorObj, mdlObj, taskObj, mogObj`

Blocks

The following blocks are defined for the various MDL Objects:

MDL Object	Block	Sub-blocks
dataObj	`DECLARED_VARIABLES`	
	`DATA_INPUT_VARIABLES`	
	`SOURCE`	
	`DATA_DERIVED_VARIABLES`	
designObj	`DECLARED_VARIABLES`	
	`INTERVENTION`	
	`SAMPLING`	
	`DESIGN_PARAMETERS`	
	`POPULATION`	
	`STUDY_DESIGN`	
	`DESIGN_SPACES`	
parObj	`STRUCTURAL`	
	`VARIABILITY`	
priorObj	`[PRIOR_PARAMETERS]`	
	`NON_CANONICAL_DISTRIBUTION`	
	`PRIOR_VARIABLE_DEFINITION`	

MDL Object	Block	Sub-blocks
mdlObj	`IDV`	
	`VARIABILITY_LEVELS`	
	`COVARIATES`	
	`FUNCTIONS`	
	`POPULATION_PARAMETERS`	
	`STRUCTURAL_PARAMETERS`	
	`VARIABILITY_PARAMETERS`	
	`GROUP_VARIABLES`	
	`RANDOM_VARIABLE_DEFINITION`	
	`INDIVIDUAL_VARIABLES`	
	`MODEL_PREDICTION`	
		`DEQ`
		`COMPARTMENT`
	`OBSERVATION`	
taskObj	`ESTIMATE`	
	[`SIMULATE`]	
	[`EVALUATE`]	
	[`OPTIMISE`]	
mogObj	`INFO`	
	`OBJECTS`	

Functions

Mathematical functions are provided to support definition of models. The following mathematical functions are defined:

Statistical functions:

`mean, median, probit, logit, invLogit, invProbit`

Arithmetic functions:

`log, log2, log10, ln, factorial, lnFactorial, floor, ceiling, min, max,`
`abs, exp, sqrt, sum, toInt, seq, seqby, dseq, rep`

Trigonometric functions:

`sin, cos, tan, sinh, cosh, tanh, asin, acos, atan, asinh, acosh, atanh`

vector and matrix handling:

`toMatrixByRow, toMatrixByCol, asVector, inverse, triangle, transpose,`
`diagonal, gInv, det, eigen, chol, matrix`

Interpolation functions:

`linearInterp, constInterp, lastValueInterp, nearestInterp, cubicInterp,`
`pchipInterp, splineInterp`

Distributions (see below).

Distributions

Distributions are denoted using the ~ prefix. The following distributions are defined:

Name	Return Type	Argument name	Argument Types	Comment
Normal	Pdf	mean	Real	Maps to ProbOnto distribution Normal1
		sd	Real	
Normal	Pdf	mean	Real	Maps to ProbOnto distribution Normal2
		var	Real	

The following ProbOnto distributions are defined:

Name	Return Type	Argument name	Argument Types	Comment
Normal1	Pdf	mean	Real	Note that ProbOnto argument is stdev not sd
		stdev	Real	
Normal2	Pdf	mean	Real	
		var	Real	
Normal3	Pdf	mean	Real	For use with BUGS
		precision	Real	
LogNormal1	Pdf	meanLog	Real	Values given on the ln scale
		stdevLog	Real	
LogNormal2	Pdf	meanLog	Real	Values given on the ln scale
		varLog	Real	
LogNormal3	Pdf	median	Real	median on the natural scale, standard deviation on the ln scale
		stdevLog	Real	
Bernoulli1	Pmf	probability	Real	Probability argument specifies the probability of the **second** category.
Poisson1	Pmf	rate	Real	
Binomial1	Pmf	probability	Real	Probability of success (second category).
		numberOfTrials	Real	
Beta1	Pdf	alpha	Real	
		beta	Real	
Gamma1	Pdf	shape	Real	
		scale	Real	
Gamma2	Pdf	shape	Real	
		rate	Real	
InverseGamma1	Pdf	shape	Real	
		scale	Real	
NonParametric	Pdf	bins	Vector	
		probability	Vector	
MultiNonParametric	vector Pdf	bins	Matrix	
		probability	Real	
Empirical	Pdf	data	Vector	
MultiEmpirical	vector Pdf	data	Matrix	
MultivariateNormal1	vector Pdf	mean	Vector	
		covarianceMatrix	Matrix	
MultivariateNormal2	vector Pdf	mean	Vector	
		precisionMatrix	Matrix	

Name	Return Type	Argument name	Argument Types	Comment
MultivariateStudentT1	vector Pdf	mean	Vector	
		covarianceMatrix	Matrix	
		degreesOfFreedom	Real	
MultivariateStudentT2	vector Pdf	mean	Vector	
		precisionMatrix	Matrix	
		degreesOfFreedom	Real	
NegativeBinomial2	Pdf	rate	Real	
		overdispersion	Real	
StudentT1	Pdf	degreesOfFreedom	Real	
StudentT2	Pdf	mean	Real	
		scale	Real	
		degreesOfFreedom	Real	
Uniform1	Pdf	minimum	Real	
		maximum	Real	
Wishart1	Matrix Pdf	scaleMatrix	Matrix	
		degreesOfFreedom	Real	
Wishart2	Matrix Pdf	inverseScaleMatrix	Matrix	
		degreesOfFreedom	Real	
InverseWishart1	Matrix Pdf	scaleMatrix	Matrix	
		degreesOfFreedom	Real	
CategoricalNonordered1	Pmf	categoryProb	Vector	
CategoricalOrdered1	Pmf	categoryProb	Vector	
MixtureDistribution1	Pdf	weight	Vector	
		distributions	Vector of PDF	
ZeroInflatedPoisson1	Pdf	rate	Real	
		probabilityOfZero	Real	

Chapter 10

Interoperability Guide

On interoperability

A key goal of the DDMoRe project is to have an intoperability framework in which models are written in a consistent language, translated to PharmML and from there converted to target software code. Before the DDMoRe project no existing language standard existed across target software used in pharmacometrics modelling, and while the underlying models could be expressed consistently in mathematical and statistical terms, the implementation of any given model varied by tool and by user according to their experience with a given target software tool.

There is some flexibility within MDL around how the user can express the mathematical and statistical models. Having flexibility allows the user to encode models quickly in a common language (MDL) which can then be shared with others and mutually understood. This flexibility also facilitates encoding in a given target when that language construct does not have a parallel in other tools. However, we STRONGLY encourage the user to encode the majority of models in a way that will facilitate interoperability. There are MDL constructs that facilitate interoperability – these generally appear as built-in functions which translate to specific constructs in PharmML and the target software. These constructs cover many typical models and are designed to allow the user to generate code quickly and have high confidence that it will be interoperable across tools.

The Model Description Language Interactive Development Environment (MDL-IDE) should assist the user in ensuring that the models encoded are valid MDL (and as a consequence, also valid PharmML). Not all models will result in code which can be readily converted to all target tools.

These interoperability constructs will be highlighted in the subsequent sections, but users should pay particular attention to sections on the use of `GROUP_VARIABLES`, `INDIVIDUAL_VARIABLES` and the `MODEL_PREDICTION`.

Dataset conventions

There are a number of conventions in preparing data for use in MDL and for target software.

- It is assumed that the `SOURCE` data file will be present as an ASCII comma-delimited text file (.csv).
- The data file should have a header row with names matching those in the `DATA_INPUT_VARIABLES` block.
- Data values should be numeric.
- Data columns with string or date:time values should have `use is ignore` for MDL.

- Null or missing values should be denoted by `.`.

Generally speaking, MDL follows NONMEM dataset conventions.

In addition the following restrictions should be observed for interoperability reasons:

- A column with `use is id` is mandatory. Values should be positive, non-zero integer, unique and contiguous.
- A column with `use is idv` is mandatory. Values should be positive, real. When the model is expressed using `DEQ` or [`COMPARTMENTS`] block the values must be monotonic increasing within ID. This constraint does not apply to analytical models. The first idv value is taken as the initial time for the model. The initial value does not need to be the same for all individuals, but it must not be lower than that of the first individual. date:time format is not supported for time.
- A column with `use is dv` is mandatory. This column can be any real value. This must have a null value for dosing records.
- When modelling multiple outcomes a column with `use is dvid` is required. Values should be positive, integer. Values should not be null for `OBSERVATION` records.
- A column with `use is mdv` is optional. Valid values are 0 (observed),1 (missing). When the `OBSERVATION` is null or missing this column should have the value 1. It can take the value 1 when `OBSERVATION`s are present if this `OBSERVATION` is to be ignored.
- A column with `use is evid` is optional. Valid values are 0 (`OBSERVATION` record), 1 (dosing record), 4 (reset and dose record).
- A column with `use is amt` is optional. For dosing records this column must be have positive, real value.
- A column with `use is rate` is optional. This column must have positive, real values. This column can only be used in combination with a column with `use is amt`. If the value is zero then a bolus dose is assumed.
- A column with `use is addl` is optional. This column must have positive, real values. This column can only be used in combination with columns with `use is amt` and `use is ii`.
- A column with `use is ii` is optional. This column must have positive, real values. This column can only be used in combination with columns with `use is amt` and `use is addl` or `use is ss`.
- A column with `use is ss` is optional. Valid values are 0 (not at steady state), 1 (at steady state). This column can only be used in combination with columns with `use is amt` and `use is ii`.
- A column with `use is cmt` is optional unless there is more than one route of administration. This column must have positive, integer values. Values in this column should start at 1 and correspond to the order of ODEs specified in the `DEQ` block of the Model Object.
- Columns with `use is covariate`, `use is catCov` and `use is variable` are optional. These columns must not have missing values. Columns with `use is catCov` must have integer values. Covariate names in the `DATA_INPUT_VARIABLES` block must match the same name (including matching case) as the header name in the source file .csv. Please see sections 2.2.5, 4.3 and 4.9 for details on the use of these variables. If the column has `use is covariate` or `use is catCov` but this variable is not declared in the `COVARIATES` block of the Model Object then it will be `dropped` and ignored.
- Columns with `use is catCov` can only have values 0,1. This implies that categorical `COVARIATES` with k values should be converted to k-1 indicator variables.
- A column with `use is varLevel` is optional. This column should not have missing values. Columns with this type should not have an underscore in the column name. The change in value of this variable denotes when to sample new values of the random variable.

Multiple uses of dataset columns

Dataset columns cannot have multiple uses defined in `DATA_INPUT_VARIABLES`. The `DATA_DERIVED_VARIABLES` block can be used to specify additional uses for dataset variables. In the current MDL, scope for using `DATA_DERIVED_VARIABLES` is limited.

For example, if the user wants to specify different outcomes / `OBSERVATION`s conditional on a dataset variable like CMT, i.e. using CMT as DVID then they will need to create a dataset variable DVID mapping into CMT values appropriately.

Defining constants in the model

For interoperability, constant values in the model should be defined as [`STUCTURAL_PARAMETERS`] and fixed to a value in the Parameter Object.

For models expressed as systems of differential equations (`DEQ` block), model parameters can be set to constant values in the `MODEL_PREDICTION` block, but this may be inefficient in the target software translation.

In the current SEE, to ensure interoperability, model parameters should not be assigned a constant value in the `GROUP_VARIABLES` or `INDIVIDUAL_VARIABLES` block.

## Interoperability in the `MODEL_PREDICTION` block

To ensure Monolix interoperability, any variable used in the `MODEL_PREDICTION` block must be either:

- the independent variable
- defined in `MODEL_PREDICTION`
- declared in `INDIVIDUAL_VARIABLES` using {type is linear, … }
- defined as `use is variable` in the `DATA_INPUT_VARIABLES` block of the Data Object

This implies in particular that `STRUCTURAL_PARAMETERS`, `VARIABILITY_PARAMETERS`, `GROUP_VARIABLES` and random variables defined in `RANDOM_VARIABLE_DEFINITION` cannot be used in `MODEL_PREDICTION`.

## Interoperability in the `OBSERVATION` block

For Monolix interoperability, different `OBSERVATION`s / outcomes must not share `VARIABILITY_PARAMETERS` and `RANDOM_VARIABLE_DEFINITION`.

For interoperability with Monolix, the residual error(s) $\sigma^2_{i,j}$ must be Normal(0,1) random variables.

Chapter 11

Changelog

v1.0 - 19 August 2016

- Public release of MDL, PharmML, DDMoRe Interoperability Framework Standalone Execution Environment (SEE)
- MDL v1.0 MDL User Guide for public release

The MDL Documented in the MDL User Guide hosted on Github is the latest release (currently v1.0) but the documentation has been updated where necessary. The version linked above is a snapshot of documentation for the IMI DDMoRe project deliverable at the close of the project.

- PharmML v0.8.1

Scope

The primary focus of MDL in this release is translation to valid PharmML, rather than conversion to target software. The previous release was primarily concerned with demonstrating interoperability across key software targets. In this version of MDL there may be MDL features which are not supported in conversion from PharmML to certain target software, but which are valid for model description and which generate valid PharmML. The aim is to widen the scope of models which can be encoded in MDL and generate PharmML for uploading to the DDMoRe repository and for future interoperability. Translation of these models to target software will follow with updates to the interoperability framework converters.

The changes to MDL since draft 7 (v0.7) enable integration of the Prior and Design Objects and improved validation of MDL giving increased confidence in generation of valid PharmML. In order to facilitate this, certain changes to syntax have been made that are ***NOT*** backwards compatible. This is regrettable since it means that existing models required changes. We do not make these changes lightly.

CHANGES v0.7 to v1.0

The follow are **critical** changes which ***break backward compatibility*** with v0.7:

Data Object

- `DECLARED_VARIABLES` block must now have type assigned to variables to enable validation of variable types between MDL objects, particularly Design Object variables.

Parameter Object

- Correlations and covariances between parameters are now specified in the Model Object and must be named parameters. This is to facilitate specification of priors on these parameters.
- Users should not specify the type of variability definition (`type is sd`, `type is var`, `type is corr`, `type is cov`) for `VARIABILITY` parameters.The variability, covariance or correlation type is specified and used in the `RANDOM_VARIABLE_DEFINITION` block in the Model Object where these parameters are defined.

Model Object

- Left hand side transformations for [`INDIVIDUAL_PARAMETERS`] are no longer valid. These were felt to be confusing.
- Right hand side functions for [`INDIVIDUAL_PARAMETERS`] and `OBSERVATION` definitions are now list definitions with the matching type to the function. This, combined with conditional statements allows more flexibility in parameter and `OBSERVATION` definitions.
- Non-continuous outcomes (binary, count, categorical) must be defined as RANDOM_VARIABLE_DEFINITION(level=D ... } and then the variable defined assigned as an anonymous list in the `OBSERVATION` block. This is to ensure that outcome variables are always defined with variability at the DV level. (This is implicit in continuous outcomes due to definition of residual error at the DV level).
- Arbitrary equations defining the outcome variable are not allowed in the `OBSERVATION` block. Use “type is userDefined” instead.

NEW FEATURES in v1.0

General

- Support for vectors and matrices
- Support for conditional assignment to lists.

Data Object

- Support for model input variables passed from the Data Object with “use is variable”. This equates to “regressor” type inputs to models.
- Support for `DATA_DERIVED_VARIABLES` where dose amounts and dose times can be derived from data columns which are being used otherwise as `use is amt` or `use is idv`.

Parameter Object

No new features. See changes section above.

Model Object

- Support for definition of parameters in `RANDOM_VARIABLE_DEFINITION` i.e. definition of CL ∼ Normal(mean=POP_CL, sd=PPV_CL) for subsequent use in [`INDIVIDUAL_PARAMETERS`]. This combined with support for ProbOnto definitions allows the user to define multivariate distributions of parameters, mixture distributions etc.

- Support for ProbOnto distributions in `RANDOM_VARIABLE_DEFINITION`.
- Support for combination of compartment definitions with differential equations.
- Support for userDefined specification of the relationship between model predictions and residual error random variables in the `OBSERVATION` block.

Task Properties object

- Support for target software specific Task Properties object.

MOG Object

- Support for specifying problem statement and model name in the `INFO` block.

Prior Object

The Prior Object is a new block for v1.0

Design Object

The Design Object is a new block for v1.0

v0.7 - 11 December 2015

- First public release of MDL
- MDL v0.7 MDL User Guide
- PharmML v0.6

Chapter 12

References

Bauer, R. 2011. *NONMEM Users Guide Introduction to Nonmem 7.2. 0. ICON Development Solutions Ellicott City, MD.* Ellicott City, MD: Icon Development Solutions.

Bazzoli, Caroline, Sylvie Retout, and France Mentré. 2010. “Design Evaluation and Optimisation in Multiple Response Nonlinear Mixed Effect Models: PFIM 3.0.” *Computer Methods and Programs in Biomedicine* 98 (1). Elsevier BV: 55–65. doi:10.1016/j.cmpb.2009.09.012.

Foracchia, Marco, Andrew Hooker, Paolo Vicini, and Alfredo Ruggeri. 2004. “Poped, a Software for Optimal Experiment Design in Population Kinetics.” *Computer Methods and Programs in Biomedicine* 74 (1). Elsevier BV: 29–46. doi:10.1016/s0169-2607(03)00073-7.

Gelman, Andrew. 2006. “Prior Distributions for Variance Parameters in Hierarchical Models(Comment on Article by Browne and Draper).” *Bayesian Analysis* 1. Institute of Mathematical Statistics: 515–34. doi:10.1214/06-ba117a.

Harnisch, L, I Matthews, J Chard, and M O Karlsson. 2013. “Drug and Disease Model Resources: A Consortium to Create Standards and Tools to Enhance Model-Based Drug Development.” *CPT: Pharmacometrics & Systems Pharmacology* 2 (3). Wiley-Blackwell: e34. doi:10.1038/psp.2013.10.

Holford, Nick. 2013. “A Time to Event Tutorial for Pharmacometricians.” *CPT: Pharmacometrics & Systems Pharmacology* 2 (5). Wiley-Blackwell: e43. doi:10.1038/psp.2013.18.

Jonsson, E.Niclas, and Mats O Karlsson. 1998. “Xposean S-PLUS Based Population Pharmacokinetic/Pharmacodynamic Model Building Aid for NONMEM.” *Computer Methods and Programs in Biomedicine* 58 (1). Elsevier BV: 51–64. doi:10.1016/s0169-2607(98)00067-4.

Lavielle, M. 2012. *MONOLIX User Guide. Lixoft, Orsay, France.* Antony, France; Inria, Orsay, France: Monolix, Lixoft. http://www.lixoft.eu/.

Lavielle, Marc. n.d. *Simulx.* Orsay, France: INRIA Xpop. http://simulx.webpopix.org/userguide/.

Lavielle, Nick Holford; Marc. 2011. “A Tutorial on Time to Event Analysis for Mixed Effect Modellers.” http://www.page-meeting.org/pdf_assets/2573-time-to-event-tutorial.pdf.

Lunn, David J., Andrew Thomas, Nicky Best, and David Spiegelhalter. 2000. “WinBUGS - a Bayesian Modelling Framework: Concepts, Structure, and Extensibility.” *Statistics and Computing* 10 (4). Springer Nature: 325–37. doi:10.1023/a:1008929526011.

MATLAB. 2000. Natick, MA: The MathWorks Inc. http://nl.mathworks.com/products/matlab.

Swat, Maciej J., Pierre Grenon, and Sarala Wimalaratne. 2016. “ProbOnto: Ontology and Knowledge Base of Probability Distributions.” *Bioinformatics* 32 (17). Oxford University Press (OUP): 2719–21.

doi:10.1093/bioinformatics/btw170.

Swat, MJ, S Moodie, SM Wimalaratne, NR Kristensen, M Lavielle, A Mari, P Magni, et al. 2015. “Pharmacometrics Markup Language (PharmML): Opening New Perspectives for Model Exchange in Drug Development.” *CPT: Pharmacometrics & Systems Pharmacology* 4 (6). Wiley-Blackwell: 316–19. doi:10.1002/psp4.57.