MODEL DESCRIPTION LANGUAGE (MDL)

USER GUIDE AND REFERENCE MANUAL

MDL Development Team

MDL Version 7.0

December 2015

MDL Development Team

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

- Nick Holford (Uppsala University) – MDL lead, pharmacometric concepts, pharmacology, modelling and simulation tasks, use case development, NONMEM, NM-TRAN to MDL conversion.
- Mike K Smith (Pfizer) – ddmore R package lead, MDL developer, interoperability, model-based drug development, Pharma industry perspective, statistical concepts, use case development, grammar.
- Natallia Kokash (Leiden University) – Implementation, grammar, MDL-IDE, Eclipse, Xtext, conversion to PharmML.
- Stuart Moodie (Eight Pillars Ltd / Pfizer) – Implementation, grammar, MDL-IDE, Eclipse, Xtext, Xtend, 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 (Universite di Pavia) – Prior Object, Bayesian, BUGS task specification.
- Zinnia Parra-Guillen (Freie Universität Berlin) – Training material, documentation, Use Case specification and testing.
- Niels Rode-Kristensen (Novo Nordisk) – Interoperability lead
- Richard Kaye (Mango Solutions) – Integration Framework technical lead, SEE builds
- Gareth Smith (Cyprotex) – Common converter
- Henrik Nyberg (Mango Solutions / Uppsala University) – PharmML to NM-TRAN translation
- Eric Blaudez (Lixoft) – PharmML to MLXTRAN conversion.
- Nadia Terranova (Merck) – Standard Output object definition
- Rikard Nordgren (Uppsala University) – NONMEM to Standard Output object translation
- 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. In alphabetical order:

- *Roberto Bizzotto (Consiglio Nazionale delle Ricerche)*
- *Elisa Borella (Uni di Pavia)*
- *Letizia Carrara (Uni di Pavia)*
- *Phylinda Chan (Pfizer)*
- *Marylore Chenel (Servier)*
- *Ronald Gieschke (Roche)*
- *Lutz Harnisch (Pfizer)*
- *Niklas Hartung (Freie Universität Berlin)*
- *Andrew Hooker (Uppsala University)*
- *Mats Karlsson (Uppsala University)*
- *Charlotte Kloft (Freie Universität Berlin)*
- *Marc Lavielle (INRIA)*
- *Andrea Mari (Consiglio Nazionale delle Ricerche)*
- *France Mentre (INSERM)*
- *Lorenzo Pasotti (Uni di Pavia)*

DDMoRe

www.ddmore.eu

Report bugs, problems with MDL, ddmore R package, MDL-IDE, Integration Framework (SEE)
http://www.ddmore.eu/forum

FAQ about DDMoRe
http://www.ddmore.eu/faq

Download MDL-IDE software:
https://sourceforge.net/projects/ddmore/files/install/SEE/

PharmML
http://www.pharmml.org/

Contents

1 Introduction

The **M**odel **D**escription **L**anguage (MDL) and the **Pharm**acometrics **M**arkup Language standard (PharmML[1]) 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[2]). 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. 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.

1.1 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)[3], BUGS[4] and MATLAB[5]. 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.

---

[1] Swat, MJ et al (2015) Pharmacometrics Markup Language (PharmML): Opening New Perspectives for Model Exchange in Drug Development. CPT: Pharmacometrics & Systems Pharmacology, 4: 316–319. doi: 10.1002/psp4.57

[2] Beal S, Sheiner LB, Boeckmann A, & Bauer RJ, NONMEM User's Guides. (1989-2009), Icon Development Solutions, Ellicott City, MD, USA, 2009.

[3] Monolix, Lixoft, Antony, France and Inria, Orsay, France. http://www.lixoft.eu/

[4] Lunn DJ, Thomas A, Best N & Spiegelhalter D (2000) WinBUGS -- a Bayesian modelling framework: concepts, structure, and extensibility. Statistics and Computing, 10:325--337

[5] MATLAB, The MathWorks Inc., Natick, MA http://nl.mathworks.com/products/matlab

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.[6]

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).7

1.2 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.

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.

1.3 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.

1.4 MDL components and structure

The MDL objects are typically defined in a file with extension .mdl. Models will typically 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.

---

[6] Harnisch L, Matthews I, Chard J and Karlsson MO (2013), Drug and Disease Model Resources: A Consortium to Create Standards and Tools to Enhance Model-Based Drug Development. CPT: Pharmacometrics & Systems Pharmacology, 2: 1–3. doi: 10.1038/psp.2013.10

An overview of the currently specified MDL objects is shown in Figure 1.

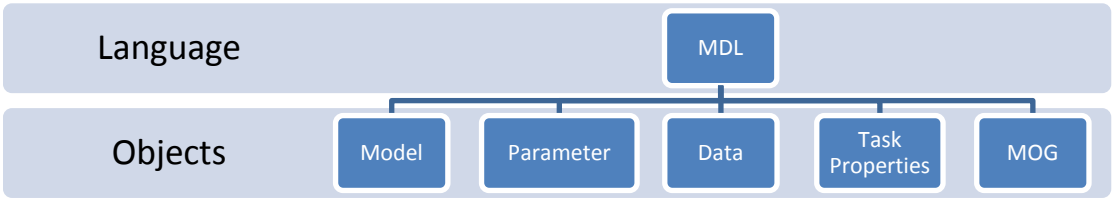

Figure 1 MDL Objects

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). Future versions of MDL will add Design and Prior Objects to extend functionality.

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 or simulation.

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 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 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, parameter values from the Parameter 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 and simulation task.

Objects are defined and stored in a MDL file with extension .mdl. It is possible to define more than one Model, Data, Parameterand 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 in a MDL file used for a task.

1.5 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. 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)[7]. 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[8], and mlxR[9] R packages. The ddmore R package will extend and enhance what the analyst can do with existing M & S 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).

1.6 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 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.

1.7 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

---

[7] Lindbom L, Ribbing J, Jonsson EN, Perl-speaks-NONMEM (PsN)—a Perl module for NONMEM related programming, Computer Methods and Programs in Biomedicine, Volume 75, Issue 2, August 2004, Pages 85-94

[8] Jonsson EN & Karlsson MO (1999) Xpose--an S-PLUS based population pharmacokinetic/pharmacodynamic model building aid for NONMEM. Computer Methods and Programs in Biomedicine. 58(1):51-64.

[9] INRIA POPIX team. http://simulx.webpopix.org/

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.

1.8 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` defining fixed effects variables, `INDIVIDUAL_VARIABLES` defining the relationship between data-defined variables, `GROUP_VARIABLES` and random effects and the `MODEL_PREDICTION` where these variables are used to calculate predictions.

1.9 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.

Balancing the priorities of interoperability and flexibility has led to an evolution of MDL from its initial form to what you see today. It will evolve further – with increased focus on flexibility – to allow modellers to convey the structure of their models in a language that is consistent and clear.

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.

2 The 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.

2.1 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 "legacy" 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 deprecated 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.

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.

2.2 DECLARED_VARIABLES Block

This block links variables defined in the Model Object with variables defined within the Data Object - occasionally we need to refer to Model Object variables while describing the data constructs. Since the

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

For example: When defining the Pharmacokinetic model analytically we may use a dosing variable D, and an observation variable Y in the Model Object. Within the Data Object we need to refer to the Model Object dosing variable D in defining the dosing variable, specified with "**use** is amt" (see section 1.3 below). In defining the Data Object dataset column containing the dependent variable (DV) specified with "**use** is dv" we need to refer to its corresponding observation variable in the Model Object Y.

The declared variables block would then be

DECLARED_VARIABLES{ D Y }

Note that the 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 OBSERVATION block variable Y.

2.3 DATA_INPUT_VARIABLES Block

This block defines the columns in the dataset described in **SOURCE** block and how these map to model variables. Columns in the data which are not required in the model should have "**use** is ignore".

- 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.***
- **DATA_INPUT_VARIABLES** cannot have more than one **use** defined.
- All **DATA_INPUT_VARIABLES** must have a **use** defined.

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 synonymous with TIME.
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 dv. . Defines the observation level of variability other than individual and observation level. Typically used to define variance levels such as interoccasion variability.
covariate	Covariate for use in definition of fixed effect variables.
catCov	Categorical covariate for use in definition of fixed effect variables based on categories.
dvid	Dependent variable identifier fwhen there is more than one type of

	observation.
mdv	Missing dependent variable
cmt	Compartment identifier
ss	Steady State indicator
ii	Inter-dose interval i.e. time between repeated doses
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, covariate corresponds to any NONMEM data item without a **use** defined by its name.

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.

However, within the current Standalone Execution Environment (SEE) interoperability framework, certain names for DATA_INPUT_VARIABLES are required to facilitate translation to NONMEM, Monolix and on to downstream R packages such as Xpose. It is expected that future versions will relax these constraints. The variables under constraint are AMT, DOSE, TIME and DV. See below for more information.

2.3.1 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 a single, specified model variable.

The syntax for defining the dose amount is:

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

For example:

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

For the current version of the interoperability framework SEE, the name of this variable must be AMT or DOSE.

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" 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 may be necessary to declare the dosing variable within the MODEL_PREDICTION block:

```
MODEL_PREDICTION {
```

```
    D # 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)) )
} # end MODEL_PREDICTION
```

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 2.3.6 and 2.3.7 on assignment using "**variable** = …" compared to using "**define** = …".

2.3.2 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:

```
TIME : { use is idv }
```

For the current version of the SEE, the name of the independent variable must be TIME.

2.3.3 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 is mapped to the Model Object **OBSERVATION** block prediction variable.

For the current version of the interoperability framework SEE, the name of the dependent variable must be DV.

2.3.3.1 Continuous data, single Model Object OBSERVATION 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 **OBSERVATION** block. The Model Object **OBSERVATION** block prediction variable name must be declared in the Data Object **DECLARED_VARIABLES** block.

It is possible to map the dependent variable to a single, specified Model Object **OBSERVATION** block variable using "**variable** = <NAME>":

For example:

```
DECLARED_VARIABLES{ Y }

DATA_INPUT_VARIABLES{

…

  DV : { use is dv, variable = Y }

…
}
```

2.3.3.2 Multiple Model Object OBSERVATION predictions

If mapping the single dataset dependent variable column (with "**use** is dv") to multiple Model Object **OBSERVATION** variables, it is necessary to also define a **DATA_INPUT_VARIABLE** with "**use** is dvid". The user must also define how to map the Model Object **OBSERVATION** block variables to values in the **DATA_INPUT_VARIABLE** with "**use** is dvid".

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

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

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 **OBSERVATION** block variable CP_obs, and when this variable has the value 2 it will be mapped to PCA_obs.

2.3.4 Defining variability levels

The model hierarchy (levels of variability) are defined within the Model Object in the **VARIABILITY_LEVELS** block. 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". Note that occasionally, if modelling summary level data in a model-based meta-analysis, the "individual" defined in the data may be different, for example, the treatment arm.

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 }
```

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.

2.3.5 Defining covariates

2.3.5.1 Defining continuous covariates and regressors

Continuous covariates are defined as "**use** is covariate".

Note that when variables are defined within the Model Object **COVARIATES** block and used in the specification of **INDIVIDUAL_VARIABLES** 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.
- Only simple transformations within the Model Object **COVARIATES** block are allowed e.g. centering on a median / mean and/or log transformation, logit transformation.
- The transformation cannot depend on another covariate or a model parameter.
- Statistical models (random effects) on covariates are not supported.

In the current version of the interoperability framework SEE, 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.

Certain target software make a distinction between covariates as defined above and "regressors" which may be used to compute variables in the Model Object **GROUP_VARIABLES** or **MODEL_PREDICTION**

blocks. Regressors do not have to follow the rules above. The current version of MDL does not make this distinction explicitly, but it is made implicitly when a variable with "**use** is **cov**" is used within the "linear" definition function within the **INDIVIDUAL_VARIABLES** block. Any variable associated with the **cov** attribute of **fixEff** within that function must have the properties described above. Users should be aware of this when defining the statistical and structural model and ensure that covariates used in this way conform to the stated rules to ensure interoperability between software. Further discussion of this point is made in the Model Object section of this User Guide.

2.3.5.2 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. The mapping is performed by using the keyword "withCategories" and then specifying the mapping between categories and values using <category> when <value> 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 **COVARIATES** 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] }

To workaround this, 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}

and then use food.missing in a piecewise assignment within the Model Object to explicitly handle the missing case. For example:

```
COVARIATES{
      food withCategories { fed, fasted, missing }
}

GROUP_VARIABLES{
      FOOD_EFFECT = if(food == food.fed) then POP_FCL_FOOD
            elseif (food== food.fasted || food == food.missing) then 0
}
```

2.3.6 Assignment to a single variable using "variable = <NAME>"

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 Y }

DATA_INPUT_VARIABLES{

 …

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

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

2.3.7 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_variable>, etc.}. The Model Object variables used in this definition must also be declared in the **DECLARED_VARIABLES** block.

```
DECLARED_VARIABLES{ CP_obs PCA_obs }

DATA_INPUT_VARIABLES{

  …

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

  …

}
```

2.4 SOURCE Block

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 MDL syntax is as follows:

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

For example:

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

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

3 The 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.

Note that as with other Objects, the Parameter Object must be self-contained – variables defined in the Model Object which are required for defining parameters in the Parameter Object must be declared in the **DECLARED_VARIABLES** block.

3.1 STRUCTURAL Block

The **STRUCTURAL** block defines the numerical values of the structural parameters with optional constraints (lo 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 (Note: no quotation marks). 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
```

3.1.1 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.

3.2 VARIABILITY Block

The **VARIABILITY** block defines the names and values of random effect 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.

The **VARIABILITY** block has a more complex structure because it needs to express both values for variance parameters, but also any correlations or covariances between these variance parameters (random effects).

Currently, MDL supports definition of the VARIABILITY (random effect) parameters and definition of the covariance or correlation between these parameters (see section 3.2.1).

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. **The parameter value type must correspond to the type used in the Model Object RANDOM_VARIABLE_DEFINITION block.**

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 }
       PPV_KA : { value = 0.1, type is sd }
       PPV_TLAG : { value = 0.1, type is sd, fix=true }
       } # end VARIABILITY
```

3.2.1 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.

Note that value expects a vector, so even if a single correlation is specified (one value) then this must be enclosed in square brackets to signify this is a vector with one element.

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

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

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.

If any VARIABILITY block variable associated with the correlation or covariance list definition has the attribute fix=true, then all diagonal and off-diagonal elements of the standard deviation - correlation matrix will be assumed to have the attribute fix=true.

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.

In the current version of MDL, if a covariance or correlation is specified between three or more parameters then all elements of that covariance or correlation must be estimated.

4 The 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**, **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 (with residual variance).

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 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.

4.1 On interoperability

The 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 any given target tool.

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 4.8, 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.

4.2 IDV Block

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 default for the independent variable is "T". This is automatically mapped to the TIME variable in the Data Object if this is specified as "`use` is `idv`".

The syntax is a simple variable declaration:

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

Note that the independent variable defined in the model will be mapped to the Data Object `DATA_INPUT_VARIABLES` variable with "`use` is `idv`".

4.3 COVARIATES Block

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. 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 (or be a subset) or those specified for DATA_INPUT_VARIABLES with "`use` is `catCov`" – see the definition of the SEX covariate above.

As discussed previously in section 3.3.5 (and reiterated here), only covariates with the following properties can be used to define linear (after transformation) relationships between population parameters, covariates and random effects:

- **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 equation cannot depend on another covariate.**
- **Statistical models (random effects) on covariates are not supported.**

Certain target software make a distinction between covariates as defined above and "regressors" which are typically used as in **GROUP_VARIABLES** block definitions or as inputs to the **MODEL_PREDICTION** block. Regressors do not have to follow the rules above. Definition of a covariate as regressor is currently handled in conversion to target software code, rather than in MDL. Regressors must have "**use** is **covariate**" in **DATA_INPUT_VARIABLES** in the Data Object and must be declared in the **COVARIATES** block in the Model Object.

Note that in the current interoperability framework SEE, there is a limit of one regressor allowed in a model. Regressors can only be used in the MODEL_PREDICTION block.

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 INDEPENDENT_VARIABLES block if WT follows the above rules for covariates.

Please also read about the specification of covariate models in sections 4.8 and 4.9.

4.4 STRUCTURAL_PARAMETERS Block

This block declares all structural and fixed effect parameters used in the model. These are parameters that will be estimated (if not fixed) or used as the basis for simulation and optimal design. As with the **DECLARED_VARIABLES** block, 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.

```
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
        RUV_PROP
        RUV_ADD
} # end STRUCTURAL_PARAMETERS
```

4.5 VARIABILITY_PARAMETERS Block

Similar to the **STRUCTURAL_PARAMETERS** block, this block declares all the variability parameters (population parameter variability and other variability level parameters) used in the model. These are parameters that will be estimated or used as the basis for simulation and optimal design. As with the **DECLARED_VARIABLES** block, 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.

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

For example:

```
VARIABILITY_PARAMETERS {
        PPV_CL
        PPV_V
        PPV_KA
        PPV_TLAG
} # end VARIABILITY_PARAMETERS
```

For reasons of interoperability, residual unexplained variability parameters should be encoded as structural parameters in MDL, which then scale random N(0,1) variates in the definition of residual error models. See section 4.11.1 for further details.

4.5.1 Residual Unexplained Variability

Residual variability is typically defined as a standard Normal distribution ~N(mean=0, var=1). The library residual error models then define the parameters of that model e.g. additive and proportional which multiply the random N(0,1) variable. As such these additive and proportional parameters are typically defined in the **STRUCTURAL_PARAMETERS** block.

4.5.2 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 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.

4.6 VARIABILITY_LEVELS Block

The **VARIABILITY_LEVELS** block defines the model hierarchy. It lists the variables defined in the DATA_INPUT_VARIABLES block with "**use** is **dv**" or "**use** is **id**" or "**use** is **varLevel**". Each variable should have attributes defining its **level** in the model hierarchy and variability **type** which is one of "**parameter**" or "**observation**".

As discussed in the **DATA_INPUT_VARIABLES** block description, we assume that level = 1 is the level of each observation. Additional levels of the hierarchy are added to 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**".

```
VARIABILITY_LEVELS{
  <Variable name> : { level = <number>,
                      type is <parameter / observation>}
```

```
}
```

For example:

```
VARIABILITY_LEVELS{
   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 a commonly used was to treat the occasion as an additional layer of inter-individual variability which changed with an occasion variable in the dataset. In MDL this is explicitly treated as a distinct level of variability.

```
VARIABILITY_LEVELS{
   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 in future versions of MDL.

4.7 RANDOM_VARIABLE_DEFINITION Block

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. The user specifies which level through the "`(level = <varLevel> )`" syntax following the `RANDOM_VARIABLE_DEFINITION` block name. Separate `RANDOM_VARIABLE_DEFINITION` blocks should be used for each layer of the model hierarchy.

The following syntax is used to define random variables

```
RANDOM_VARIABLE_DEFINITION( level = <DATA_INPUT_VARIABLE with
                                          use is id  | dv | varlevel> ){

        <VARIABLE NAME > ~ <Distribution with arguments >
}
```

The following distributions are defined

Name	Argument name	Argument Types
Normal	mean sd	Real Real
Normal	mean var	Real Real

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. **Note that in the current MDL version, the use of standard deviation or variance must match what is specified for that parameter in the Parameters Object VARIABILITY block.**

Correlations and covariances between the ETAs are defined within the Parameter Object. This allows the user to test different correlation and covariance structures between variables without having to change the Model Object. Typically the random effect variables will be Normally distributed, so covariances and correlations between variables imply a multivariate-Normal distribution.

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 model hierarchy through the **VARIABILITY_LEVELS** block.

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

And appropriate specification within the Data Object of **DATA_INPUT_VARIABLES** with "**use** is id", "**use** is dv" and "**use** is varLevel".

The current version of MDL supports only Normal distributions for random effects.

4.8 GROUP_VARIABLES Block

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 essentially arbitrary. The MDL-DE is not equipped to determine whether the defined relationships conform to linear relationships (after transformation) that have been shown to improve 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))
       GRPCL = POP_CL * FSIZE * FAGE * FMAT
}
```

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

The use of GROUP_VARIABLES to define variables will rule out interoperability of the model with Monolix. To ensure interoperability with Monolix, (simple) covariate transformations should occur in the COVARIATES block, definition of covariate relationships should be defined in the INDIVIDUAL_VARIABLES block using the linear(…) construct. Constants should be defined as fixed STRUCTURAL_PARAMETERS or defined in the MODEL_PREDICTION block.

4.8.1 On defining model constants - interoperability

Constants can be defined in several ways in MDL. How they are defined depends on the use and meaning of the value in a particular model. In general, variables assigned a constant value should be defined in the GROUP_VARIABLES block. This will ensure that they are translated appropriately in the target software code.

They can also be defined in the MODEL_PREDICTION block, ***however*** if differential equations are also defined in MODEL_PREDICTION then the variables may be grouped with differential equation code in the target software translation, which is inefficient.

Lastly they can also be defined as structural model parameters in the Parameter Object with the attribute fix = true. An example is the structural parameter defining the effect of the weight covariate WT on Volume of distribution, BETA_V_WT which is fixed to a value of 1 in the MDL Use Cases.

Variables should ***not*** be assigned a constant value in the INDIVIDUAL_VARIABLES block. They would then be treated as individual model parameters for estimation by some target software translations and in other tasks – such as simulation, prediction, model diagnostics etc.

4.9 INDIVIDUAL_VARIABLES Block

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.

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 all target software is the "linear after transformation" method described in section 4.9.1 below.

4.9.1 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:

```
h(<Individual parameter>)
  = 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) ] )
```

Note that h(<Individual parameter>) on the left hand side of the equation is a function whereas the <h> in "**trans** is <**h**>" is an enumerated type. These should be the same transformation. If a transformation is used, then the Individual parameter (left hand side of the equation) is implicitly back-transformed for use in later calculations, for example in the **MODEL_PREDICTION** block.

For example:

```
ln(CL) = 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:

```
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] )
```

If between occasion variability is specified in a **RANDOM_VARIABLE_DEFINITION** block then the associated random effects can be added into the ranEff vector. These will be added into the linear equation. For example:

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

4.9.2 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).

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

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

ψ_i – individual parameter

β – Fixed effects

C_i – Covariates

η_i – Random effect(s)

H – arbitrary function

h – transformation function – log, logit, probit.

Where $H(\beta, C_i)$ is defined in the **GROUP_VARIABLES** block. If a transformation *h(.)* e.g. ln is applied to the left hand side, then the user should ensure that the variable passed from the **GROUP_VARIABLES** block $H(\beta, C_i)$ is on the log scale.

The MDL syntax for this form of specification is:

```
h(<Individual parameter>)
  = general(
              grp = <GROUP_VARIABLES defined variable >,
              ranEff = [ RANDOM_VARIABLE_DEFINITION parameter(s) ] )
```

Note that there is no "**trans** is" function argument on the right hand side. It is assumed that appropriate transformations have been made in the **GROUP_VARIABLES** block to ensure that the fixed effect and random effect are additive and on the correct scale given the left hand side transformation.

For example, for the GROUP_VARIABLES defined in section 4.8 above:

```
ln(CL) = general(grp = ln(GRP_CL),
                 ranEff = [ETA_CL])
```

4.9.3 Mixed effect model defined by equations

The individual variables can also be defined by combining parameters or variables defined in **GROUP_VARIABLES** and random effects in equations.

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 covariate effects directly in the equation as follows:

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

Which can be log transformed into a linear 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-DE to determine this. To specify linear models we must explicitly do so using the linear(…) construct described in section 4.9.1.

4.9.4 INDIVIDUAL_VARIABLES without inter-individual variability

Any parameters defined in the **STRUCTURAL_PARAMETERS** block without associated inter-individual variability i.e. where the individual value is the (constant) population value, must be defined within the **INDIVIDUAL_VARIABLES** block.

In translation to Monolix, these parameters are assumed to be defined as real i.e. can be estimated to be negative. If the parameter should be constrained to be positive then the user can define these using a left hand side transformation and the linear(…) construct with IIV fixed to zero.

4.9.5 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 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 disease progression model:

```
EMAX = EMAX_BASELINE + EMAX_BETA*TIME + ETA_EMAX
```

It may be tempting to try to write this as a linear(…) relationship with CP as a covariate, but recall that covariates may not be time-varying, and CP would certainly break this rule.

If we encode `GRP_PD = POP_BASELINE + POP_BETA*CP` as a **GROUP_VARIABLE** and then add ETA_PD using the general(…)form then the GRP_PD is also time-varying.

However if we break the above model into components, then we can use linear(…) to express an individual baseline

```
INDIV_BASE = linear(pop = POP_BASELINE, ranEff = [ETA_PD])
```

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

```
MODEL_PREDICTION{
    PD = INDIV_BASE + POP_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.

4.10 MODEL_PREDICTION Block

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).

Variables used in the **MODEL_PREDICTION** block must be completely defined at this point. No further calculation of group or individual variables should occur in the **MODEL_PREDICTION** block.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  # 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 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 linear(…)**
- **defined as a covariate in MDL, with the restriction mentioned in the covariate block (will be a regressor in PharmML)**

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

4.10.1 DEQ Sub-block

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

<VARIABLE> : { **deriv** = <equation>, 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
```

4.10.2 On Tlag and Bioavailability

Since MDL has no reserved variable names, there is no mechanism for target software to identify Tlag and Bioavailability unless they are used in a **COMPARTMENT** sub-block definition. Otherwise Tlag and Bioavailability have no special treatment as parameters. Estimation of inter-individual variability on a Tlag parameter is not supported with FOCE or FOCEI estimation methods in NONMEM. Similarly, for bioavailability the implication is that bioavailability cannot be used to define initial amount in an equation other than the one specified by the **DATA_INPUT_VARIABLE** with "**use** is **amt**".

4.10.3 COMPARTMENT Sub-block

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.

The current version of MDL translates **COMPARTMENT** specifications into differential equations in NONMEM and to PK Macros in Monolix. Future versions may be able to identify systems of compartments and translate these to closed-form solutions in target software.

COMPARTMENTS sub-block processes are specified as lists with arguments depending on the processes being described.

4.10.3.1 Drug input & absorption

There are two **COMPARTMENT** block processes describing drug input to the system. These are `direct` and `depot`. `direct` defines bolus or zero-order dosing, 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 drug input and absorption processes.

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

(O) - Optional attribute

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

Future versions of MDL will support the NONMEM convention of negative values for RATE indicating that the infusion duration or infusion rate is to be estimated.

4.10.3.2 Distribution processes

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

Processes with drug input or drug elimination are defined using “type is compartment”.

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

For example:

```
CENTRAL : {type is compartment, modelCmt=2}
```

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, modelCmt=3, 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	modelCmt, from, kin, kout
compartment	modelCmt
effect	modelCmt, from, keq

4.10.3.3 Elimination, transfer and effect site 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, MDL must have some way of knowing that this is a valid list, and so we have introduced the notion of an "anonymous list" for this construct. We signify this through double colons at the beginning of the list:

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

For example in the one compartment model:

```
:: {type is elimination, modelCmt=2, 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, modelCmt=2, from=LATENT, to=CENTRAL, kt=K23}
```

Compartment Type	Attribute Combination
elimination	modelCmt, from, v, k
	modelCmt, from, v, cl

	modelCmt, from, vm, km
transfer	modelCmt, from, to, kt

4.11 OBSERVATION Block

The **OBSERVATION** provides the distribution of the outcome or function defining the outcome variable, using the prediction from the **MODEL_PREDICTION** block and in the case of continuous data **RANDOM_VARIABLE_DEFINITION** at the observation ("**use** is **dv**") level. This block should contain ***only*** a functional definition for continuous outcomes or definition of a distribution for the outcome. Any calculations or equations needed for this definition should be placed in the **MODEL_PREDICTION** block.

If more than one outcome is specified (via the **DATA_INPUT_VARIABLES** block defining how the "**use** is **dv**" column data is partitioned by the "**use** is **dvid**" column – see section X.Y) then we specify each outcome separately within the **OBSERVATION** block. It is not necessary to have conditional assignment of outcomes to a single outcome variable depending on the variable with "**use** is **dvid**".

In the current version of MDL, only the standard residual error functions defined below are supported. Equation based definitions of the outcomes are not supported.

For Monolix interoperability, different observation types must use different STRUCTURAL_PARAMETERS and RANDOM_VARIABLE_DEFINITIONS.

4.11.1 Continuous outcomes

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

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

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

Where

$y_{ij} = jth$ observation for subject i

h = Transformation of the outcome to ensure that the resulting function is an additive function of *f* and *g*.

f = structural model prediction from the **MODEL_PREDICTION** block.

g = functional definition of the residual error model

ψ_i = individual parameters defined in the **INDIVIDUAL_VARIABLES** block

x_{ij} = covariates and regression variables e.g. time, concentration etc.

ξ = parameters of the residual error model

ε_{ij}= residual error defined in **RANDOM_VARIABLE_DEFINITION** block

The syntax for definition of continuous outcome variables is

< OUTCOME VARIABLE NAME> = <residual error model function (see table below) with arguments>

The following residual error model functions are defined, as previously described in MDL Language Reference section X.Y and reiterated here. These combine a **MODEL_PREDICTION** variable with **RANDOM_VARIABLE_DEFINITION** variables and any associated parameters.

Name	Return Type	Argument name	Argument Types
additiveError	Real	trans (Optional) additive prediction eps	Builtin Real Real Real
proportionalError	Real	trans (Optional) proportional prediction eps	Builtin Real Real Real
combinedError1	Real	trans (Optional) additive proportional prediction eps	Builtin Real Real Real Real
combinedError2	Real	trans (Optional) additive proportional prediction eps	Builtin Real Real Real Real

combinedError1 uses a single EPS random variable while combinedError2 uses 2 EPS random variables for the residual error model.

Additional functions will be defined in future MDL versions.

As with the linear(…) and general(…) functions in the **INDIVIDUAL_VARIABLES** block, we use functions here to make explicit the relationships between predictions and residual variance terms to facilitate interoperability between target software. A more general equation form could be used, but this would not necessarily translate successfully to all target software for estimation.

4.11.2 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).

Note that for continuous data the outcome is specified as an equation <OUTCOME VARIABLE NAME> = <residual error model function> while the specification of other non-continuous types of data use a list definition <OUTCOME VARIABLE NAME> : { **type** is … }.

4.11.2.1 Count data

For count data, we have the following syntax:

<OUTCOME VARIABLE NAME> : { **type** is **count**, **distn** = Poisson(**lambda** = <VARIABLE NAME>) }

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

```
INDIVIDUAL_VARIABLES{
        ln(indiv_BASECOUNT) = linear( trans is ln, pop = POP_BASECOUNT,
                        ranEff = [eta_PPV_EVENT] )
   }

MODEL_PREDICTION{
      lnLAMBDA=ln(indiv_BASECOUNT) + POP_BETA*CP
      LAMBDA = exp(lnLAMBDA)
   }

OBSERVATION{
      Y : { type is count, distn = Poisson(lambda = LAMBDA) }
   }
```

Note in the above example that the indiv_BASECOUNT variable is specified using the linear function and a natural log transformation on both sides to ensure that indiv_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. 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.

In the current version of MDL, only the Poisson distribution can be used to specify count data. In future versions of MDL many more distributions will be available through the Prob-Onto distribution specification in PharmML[10].

4.11.3 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	$\Lambda(a,b) = \int_a^b \lambda(t)dt$

[10] Swat MJ et al. Extensions in PharmML 0.7-0.7.2, 4 September 2015

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	$P(T < t) = \int_0^t p(s)ds$

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

The MDL syntax for time to event outcomes is :

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

For example:

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

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

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. However the convention in NONMEM datasets of using MDV to identify the start of the observation period for assessing TTE cannot be used. In order to make the model and data interoperable the user must ensure that MDV is not included in the data or use "**use** is **ignore**" for MDV in the **DATA_INPUT_VARIABLES** block.

5 The 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. In the current version of MDL, the implementation of this is very limited. Some Task Properties Object settings are generic (apply across different target software) but most are specific to the intended target software.

In the current version of MDL, only the estimation algorithm property can be set.

5.1 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 set 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.

If target software specific settings are ***not*** present for a given target software then it is assumed that default settings and options should be used for specific algorithm or method within the target software.

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.

5.2 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.

5.3 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.

5.4 `ESTIMATE` Block

In the current MDL version, the only block supported is the **`ESTIMATE`** block. The syntax for this block is :

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

As stated previously, this block will be extended in future versions of MDL to capture target software specific settings and options.

6 The MOG Object

The MOG Object is where the user defines the Data, Parameters, Model and Task Properties required for a particular task e.g. estimation.

6.1 OBJECTS Block

In the current MDL version, the only block supported is the **OBJECTS** block. This 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 estimation 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.

In the case of estimation the Objects block must have 4 statements, with one each of dataObj, parObj, mdlObj, taskObj.

The syntax for statements in this block is :

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

For example:

```
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 }
	}
}
```

6.2 Mapping of variable names between MDL Objects

The current version of MDL requires that variable names in each object are consistently named. This restriction may be relaxed in future versions of MDL.

7 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.

7.1 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.

7.1.1 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, exponentiale, false, in, inf, is, pi, set, then,
when, withCategories, true
```

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

```
ordered
```

7.1.2 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.

7.1.3 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: `[0, a, b, 25.0]`
- Strings: `"a string"`
- Integers: `99, 22, 0, -1, -477`
- Real: `99.9, -0.473, 9e-2, -0.3424e5`
- Boolean: `true, false`

7.1.4 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
```

7.1.4.1 Numerical and Boolean Expression

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

binary_op := <operand> <operator> <operand>

unary_op := <operator> <operand>

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

Operator	Symbol	Left Type	Right Type
Logical AND	&&	Boolean	Boolean
Logical OR	||	Boolean	Boolean

Less than	<	Real	Real
Greater than	>	Real	Real
Less than or equal	<=	Real	Real
Greater than or equal	>=	Real	Real
Equal	==	Real	Real
Not equal	!=	Real	Real
Power	^	Real	Real
Multiplied by	*	Real	Real
Divided by	/	Real	Real
Modulo (remainder)	%	Real	Real
Add	+	Real	Real
Subtract	-	Real	Real
Negation (unary)	-		Real
Positive (unary)			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	^
Multiplicative	* / %
Additive	+ -
Relational	< > <= >=
Equality	== !=
Logical AND	&&
Logical OR	||

7.1.4.2 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:

```
<categorical variable>.<category 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
```

7.1.4.3 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:

$$f(x) = \begin{cases} -1, & x < 0 \\ 1, & \text{otherwise} \end{cases}$$

and in MDL the equivalent is a conditional expression:

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

Expressions with more than one condition are possible:

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

which is equivalent to:

$$f(x) = \begin{cases} -1, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

Equation 1

as are combinations of elseif and else:

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

Example 1

$$f(x) = \begin{cases} -1, & x < 0 \\ 1, & x > 0 \\ 0, & \text{otherwise} \end{cases}$$

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 'else' as this guarantees all conditions are covered, but in cases such as Equation 1 this clearly is not necessary so this is not required by the language.

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:

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

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.

Finally, note that it is possible to write an expression such as this:

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

It is important to understand that this expression is different to shown in Example 1: the second if/else clause is nested in the first. The equivalent piecewise function is:

$$f(x) = \begin{cases} -1, & x < 0 \\ \begin{cases} 1, & x > 0 \\ 0, & \text{otherwise} \end{cases}, & \text{otherwise} \end{cases}$$

In order to keep conditional statements simple and to improve compatibility with other modelling tools the nesting of conditional expressions like this is current prohibited in MDL.

7.1.4.4 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.

7.1.4.5 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 1.1.9.3). The details of how this is carried out is described in detail below (section 1.1.11). 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] )
```

7.1.4.6 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.

7.1.4.7 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.

7.1.5 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.

7.1.6 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.

7.1.6.1 Equation definition

This defines a variable using a notation equivalent to a mathematical equation. It can have three forms:

1. x = <expression>
   ```
   x = 2 + 5 / ln(22)
   ```
2. fn(x) = <expression>, 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.

7.1.6.2 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.

7.1.6.3 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.

7.1.6.4 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.

7.1.6.5 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 exception that the expression on the right hand side of the '~' must have a type of PDF and the variable cannot 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)

ln(ETA_CL) ~ Normal(mean = 0, sd = PPV_CL)     # invalid

ETA_CL ~ PPV_CL                                # invalid
```

The generic syntax description is as follows:

```
<ID> ~ <PDF expression>
```

7.1.6.6 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>, …] }
```

7.1.6.7 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:

```
ESTIMATION{
    set algo is saem, algo is foocei  # invalid
}
```

or

```
ESTIMATION{
    set algo is saem
```

```
    set algo is foocci  # invalid
}
```

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.

7.1.7 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.

7.1.8 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 }
    }
}
```

7.2 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.

7.2.1.1 The types

Name	Description
Int	Integer
Real	Real number
Boolean	Boolean
Enum	An enumeration type. A namespace for categories. Enumeration types do not require quotation marks (compared to strings).
Enum value	A specific enumeration defined by an enumeration type and the value held by a variable of enumeration type.
Builtin Enum	Commonly abbreviated to BE, this is a predefined set of enumerated values that are part of the MDL definition. These are usually prefixed by the 'is' keyword and are often (but not exclusively) used in lists to indicate the key attribute.
List	A data structure that associates a set of attributes with a variable. See below for details.
Pdf	A Probability density function. Usually returned by a statistical distribution function.
String	A character string
Pmf	A probability mass function type. Typically returned by discrete probability distribution functions.
Mapping	This is the type of the data mapping syntax structure.
Sublist	This is a sublist, essentially an attribute and value pair.

Vector	A one dimensional array of values of a single type.
Reference	A reference to a value. The value can be of any type
Undefined	No type. This is used internally to indicate a validation error during type checking. This is not a valid type.

7.2.1.2 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
```

7.2.1.3 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
```

7.2.1.4 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
```

7.2.1.5 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{
        c3 : { use is doseTime, idvColumn=c2, amtColumn=c1 } # invalid
        # 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.

7.2.1.6 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
att1 is ugg  # invalid
att1 = foo  # invalid
```

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.

7.2.1.7

7.2.1.8 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.

7.2.1.9 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
```

```
Gender2 == Gender1.male # invalid

Gender1 == Gender2      # invalid
```

7.2.2 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 in to be 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.

7.3 The language semantics

The language core is a syntactic framework upon which the language that the modeller writes is built upon. This section describes the elements of the language that sit upon this framework and provide the semantics (meaning) of the language.

MDL is split into Objects (Data, Parameter, Model, Task Properties, MOG), blocks within objects and different types of statements and assignments within blocks. MDL objects and blocks have been named and defined to assist the user in writing the model - identifying where various pieces of information must be defined – and to increase readability of models.

The elements of the Data, Parameter, Task Properties and MOG object may be updated using R language and functions within the ddmore R package so that tasks and workflow may be defined programmatically. For example, the MDL parameter object may be updated with results from a Standard Output object within R e.g. parameter estimates and variance-covariance matrix of the estimate.

7.3.1 Objects

Objects group together blocks pertaining to the key elements - data, parameters, model, task properties and MOG.

Example:

warfarin_PK_Compartments_dat = **dataobj** { # Blocks and statements… }

Valid types of Objects are:

```
dataObj, parObj, mdlObj, taskObj or mogObj
```

7.3.2 Blocks

The following blocks are defined for the various MDL Objects:

MDL Object	Block	Sub-blocks
dataObj	DECLARED_VARIABLES	
	DATA_INPUT_VARIABLES	
	SOURCE	
parObj	DECLARED_VARIABLES	
	STRUCTURAL	
	VARIABILITY	
mdlObj	IDV	
	VARIABILITY_LEVELS	
	COVARIATES	
	STRUCTURAL_PARAMETERS	
	VARIABILITY_PARAMETERS	
	GROUP_VARIABLES	
	RANDOM_VARIABLE_DEFINITION	
	INDIVIDUAL_VARIABLES	
	MODEL_PREDICTION	
		DEQ
		COMPARTMENT
	OBSERVATION	
taskObj	ESTIMATE	
	SIMULATE	
mogObj	OBJECTS	

Blocks can be constrained to only constrain certain types of statement. For example a DECLARED_VARIABLES block cannot contain a list definition.

Name	Min blocks	Max blocks	Min stmts	Max stmts	Permitted Statements
COVARIATES	0		0		equation definition category definition
VARIABILITY_LEVELS	0		0		equation definition
STRUCTURAL_PARAMETERS	0		0		equation definition

VARIABILITY_PARAMETERS	0		0		equation definition
RANDOM_VARIABLES	0		0		random variable definition
INIDIVIDUAL_PARAMETER_DEFINITIONS	0		0		equation definition
MODEL_PREDICTION	0		0		equation definition list definition
DEQ	0		0		equation definition list definition
COMPARTMENT	0		0		list definition anonymous list statement
OBSERVATION	0		0		equation definition list definition
GROUP_VARIABLES	0		0		equation definition
IDV	1	1	1	1	equation definition
DATA_INPUT_VARIABLES	0		1		list definition
DECLARED_VARIABLES	0		0		equation definition category definition
DATA_DERIVED_VARIABLES	0		0		list definition
SOURCE	1	1	1	1	list definition
VARIABILITY	0		0		list definition
STRUCTURAL	0		0		list definition
ESTIMATE	0	1	0		property statement
SIMULATE	0	1	0		property statement
OBJECT	1	1	4	4	list definition

7.3.3 Functions

Mathematical functions are provided to support definition of models. The following mathematical functions are defined:

Name	Return	Argument	Argument	Comment

	Type	description	Types	
log	Real	value base	Real Real	Log of *value* to the specified base.
ln	Real	value	Real	Natural log
abs	Real	value	Real	Absolute value - Ignore sign
exp	Real	value	Real	Euler's number to the power of *value*.
seq	Real Vector	begin end interval	Real Real Real	Returns a vector containing a sequence of numbers starting at *begin*, with the step size of *interval* and ending before or at *end*.
sqrt	Real	value	Real	Square root of *value*.
Factorial	Real	value	Real	*value*!
lnFactorial	Real	value	Real	ln(*value*!)
log2	Real	value	Real	log base 2
log10	Real	value	Real	log base 10
sin	Real	value	Real	Sin of *value* (in radians)
cos	Real	value	Real	Cos of *value* (in radians)
tan	Real	value	Real	Tan of *value* (in radians)
logit	Real	value	Real	Logit function, the inverse of the logistic function (invLogit).
invLogit	Real	value	Real	Logistic function
probit	Real	value	Real	Probit function.
invProbit	Real	value	Real	Inverse of the Probit function
floor	Integer	value	Real	Round *value* down to the nearest integer
ceiling	Integer	value	Real	Round to *value* up to the nearest integer
min	Real	a b	Real Real	Compare *a* and *b* and return the lowest number.
max	Real	a b	Real Real	Compare *a* and *b* and return the highest number.
sum	Real	values	Real vector	Sum all the values in the vector *values*.
mean	Real	values	Real vector	Take the mean of the all elements in vector *values*.
median	Real	values	Real vector	Take the median of the all elements in vector *values*.

Some predefined functions are provided with MDL to assist the user in defining certain objects e.g. residual error models.

Name	Return Type	Argument name	Argument Types	Comment
additiveError	Real	trans (O) additive prediction eps	BE:transType Real Real Real	Additive residual error model
proportionalError	Real	trans (O) proportional prediction eps	BE:transType Real Real Real	Proportional residual error model
combinedError1	Real	trans (0) additive proportional prediction eps	BE:transType Real Real Real Real	Combined error 1 residual error model
combinedError2	Real	trans (0) additive proportional prediction eps	BE:transType Real Real Real Real	Combined error 2 residual error model

(0) = Optional

7.3.4 Distributions

Distributions are denoted using the ~ prefix. The following distributions are defined:

Name	Return Type	Argument name	Argument Types	Comment
Normal	Pdf	mean sd	Real Real	
Normal	Pdf	mean var	Real Real	
Bernoulli	Pmf	probability	Real	
Poisson	Pmf	lambda	Real	
Gamma	Pdf	shape scale	Real Real	

7.3.5 Lists

7.3.5.1 List definitions

As described above, a list is defined by a combination of their owning block, their key attribute, and optionally, the value assigned to the key attribute. Each list also has its own type and an alternate type. A list can also be anonymous or require a symbol definition. These options are enumerated for each list below.

DATA_INPUT_VARIABLES

Key attribute: use

Type Name	Key Value	Alternate Type	Categories Defined	Named	Description
Addl	addl	None	N	Y	Additional dose. Used in steady state dosing. Requires that 'amt' and 'ss' are defined.
Amt	amt	None	N	Y	Amount of dose to be administered.
CatCovariate	catCov	None	Y	Y	Column defines a categorical covariate.
Cmt	cmt	None	N	Y	Column provides an indicator value used to select the dosing variable.
Covariate	covariate	None	N	Y	Value of continuous covariate.
Dv	dv	None	N	Y	Value of observation.
Dvid	dvid	None	N	Y	Column provides an indicator value used to select the correct observation variable.
Id	id	None	N	Y	Column provides an identifier for the individual in the study. The value should be >= 0.
Idv	idv	None	N	Y	Column specifies the independent variable of the dataset.
Ignore	ignore	None	N	Y	Column should be ignored.
InterDoseInterval	ii	None	N	Y	Provides the inter-dose interval in steady state dosing. Required that 'ss' and 'amt' columns are defined.
Mdv	mdv	None	N	Y	Column indicates whether an observation is missing. 1 indicates missing, 0 present.
Rate	rate	None	N	Y	Column value defines the rate of infusion. Requires an 'amt' column to also be defined.

SteadyState	ss	None	N	Y	Value indicates what type of steady state dosing is being applied. 0 = none, 1 = SS with reset, 2 = SS with no reset, 3 = SS with no reset of model prediction variables.
VarLevel	varLevel	None	N	Y	Column provides an indicator value for which variability level applies. The value should be an integer value >=0. The range of values is inferred from the data.

DATA_DERIVED_VARIABLES

Key attribute: use

Type Name	Key Value	Alternate Type	Categories Defined	Named	Description
DoseTime	doseTime	None	N	Y	Selects the value independent variable value from the 'idv' column when dosing is administered (as determined from the 'amt' column).

SOURCE

Key Attribute: file

Type Name	Key Value	Alternate Type	Categories Defined	Named	Description
Source		None	N	Y	Specifies the source of the data to be used by the data object.

VARIABILITY_LEVELS

Key Attribute: type

Type Name	Key Value	Alternate Type	Categories Defined	Named	Description
VarLevel		None	N	Y	Defines a variability level in the model.

COMPARTMENT

Key Attribute: type

Type Name	Key Value	Alternate Type	Categories Defined	Named	Description
Direct	direct	Real	N	Y	Defines a direct dosing input into a compartment.
Effect	effect	Real	N	Y	Defines an effect compartment.
Depot	depot	Real	N	Y	Defines depot dose administration.
Transfer	transfer	None	N	N	Defines a transfer process between 2 compartments.
Compartment	compartment	Real	N	Y	Defines a compartment.
Elimination	elimination		N	N	Defines an elimination process from a compartment.
Distribution	distribution		N	Y	Defined a peripheral compartment where drug is distributed too.

DEQ and MODEL_PREDICTION

Key Attribute: deriv

Type Name	Key Value	Alternate Type	Categories Defined	Named	Description
Deriv		Real	N	Y	Defined an ODE.

VARIABILITY

Key Attribute: type

Type Name	Key Value	Alternate Type	Categories Defined	Named	Description
CovarMatrix	cov	None	N	Y	Defines a covariance matrix.
CorrMatrix	corr	None	N	Y	Defines a correlation matrix.
SDEstimate	sd	Real	N	Y	Defines an standard deviation initial estimate.

VarEstimate	var	Real	N	Y	Defines a variance initial estimate.

STRUCTURAL

Key Attribute: value

Type Name	Key Value	Alternate Type	Categories Defined	Named	Description
StructuralEstimate		Real	N	Y	Defines an initial parameter estimate range.

OBJECTS

Key Attribute: type

Type Name	Key Value	Alternate Type	Categories Defined	Named	Description
MdlObjInMog		Real	N	Y	Defines the MDL object to be included in the MOG.

OBSERVATION

Key Attribute: type

Type Name	Key Value	Alternate Type	Categories Defined	Named	Description
CountObs	count	Real	N	Y	Defines a count observation.
TteObs	tte		N	Y	Defines a time to event observation

7.3.5.2 List Attributes

The attributes used in each list type are detailed below complete with their expected type and the expected attribute combinations. **Abbreviations: BE - builtin enumeration, M - mandatory, O optional.**

DATA_INPUT_VARIABLES

List Name	Combination	Attribute	Type	Description

Addl	use (M)	use	BE:divUse	
Amt	use (M), define (M)	use	BE:divUse	
		define	Mapping Type	Selects variable based on value in a 'cmt' column. Data value in this column is mapped to selected variable.
	use (M), variable (M)	variable	ref:Real	Maps values in column to variable.
CatCovariate	use (M)	use	BE:divUse	**categories defined using 'withCategories'**
Cmt	use (M)	use	BE:divUse	
Covariate	use (M)	use	BE:divUse	
Dv	use (M), define (M)	use	BE:divUse	
	use (M), variable (M)	define	Mapping Type	Selects variable based on value in a 'dvid column. Data value in this column is mapped to the selected variable.
		variable	ref:Real	Maps values in column to variable.
Dvid	use (M)	use	BE:divUse	
Id	use (M)	use	BE:divUse	
Idv	use (M)	use	BE:divUse	
Ignore	use (M)	use	BE:divUse	
InterDoseInterval	use (M)	use	BE:divUse	
Mdv	use (M)	use	BE:divUse	
Rate	use (M)	use	BE:divUse	
SteadyState	use (M)	use	BE:divUse	
VarLevel	use (M)	use	BE:divUse	

DATA_DERIVED_VARIABLES

List Name	Combination	Attribute	Type	Description
DoseTime	use (M), idvColumn (M),	use	BE:ddUse	

	amtColumn (M)	idvColumn	ref:Idv	The independent variable column.
		amtColumn	ref:Amt	The dose amount column.

SOURCE

List Name	Combination	Attribute	Type	Description
Source	file (M), inputFormat (M)	file	String	The location of the data file specified as a path.
		inputFormat	BE:input	The independent variable column.

VARIABILITY_LEVELS

List Name	Combination	Attribute	Type	Description
VarLevel	type (M), level (M)	type	BE:varLvlType	The type of variability, observational or parameter.
		level	Int	The level of variability startng at 1.

COMPARTMENT

List Name	Combination	Attribute	Type	Description
Direct	type (M), modelCmt (O), to (M), modelDur (O), tlag (O), finput (O)	type	BE:compType	The type of compartment macro
		modelCmt	Int	An optional number to label the macro.
		to	ref:List:Compartment	The compartment the does is administered to.
		modelDur	Real	
		tlag	Real	Time lag in start of administration.
		finput	Real	
Effect	type (M), modelCmt (O), from (M), keq (M)	type	BE:compType	The type of compartment macro
		modelCmt	Int	An optional number to label the macro.
		from	ref:List:Compartment	Compartment drug is coming from.
		keq	Real	Equilibrium constant
Depot	type (M), modelCmt (O),	type	BE:compType	The type of

	to (M), ka (M), tlag (O), finput (O) type (M), modelCmt (O), to (M), ktr (M), mtt (M) type (M), modelCmt (O), to (M), modelDur (M), tlag (O), finput (O)			compartment macro
		modelCmt	Int	An optional number to label the macro.
		to	ref:List:Compartment	The compartment the does is administered to.
		modelDur	Real	
		tlag	Real	Time lag in start of administration.
		finput	Real	
		ktr	Real	
		mtt	Real	
Transfer	type (M), modelCmt (O), from (M), to (M), kt (M)	type	BE:compType	The type of compartment macro
		modelCmt	Int	An optional number to label the macro.
		from	ref:List:Compartment	Compartment drug is coming from.
		to	ref:List:Compartment	The compartment the does is administered to.
		keq	Real	Equilibrium constant
Compartment	type (M), modelCmt (O)	type	BE:compType	The type of compartment macro
		modelCmt	Int	An optional number to label the macro.
Elimination	type (M), modelCmt (O), from (M), v (M), k (M) type (M), modelCmt (O), from (M), v (M), cl (M) type (M), modelCmt (O), from (M), vm (M), km (M)	type	BE:compType	The type of compartment macro
		modelCmt	Int	An optional number to label the macro.
		from	ref:List:Compartment	Compartment drug is coming from.
		v	Real	volume of source compartment
		cl	Real	clearance
		vm	Real	VMax
		km	Real	Michaelis Constant
Distribution	type (M), modelCmt (O), kin (M), kout (M), from (M)	type	BE:compType	The type of compartment macro
		modelCmt	Int	An optional number to label the macro.

		from	ref:List:Compartment	Compartment drug is coming from.
		kin	Real	Rate into compartment
		kout	Real	Rate out of compartment

MODEL_PREDICTION and DEQ

List Name	Combination	Attribute	Type	Description
Derivative	deriv (M), init (O), x0 (O), wrt (O)	deriv	Real	The derivative expression.
		init	Real	The initial value of derivative
		x0	Real	The initial time of the derivative.
		wrt	ref:Real	The independent variable of the derivative (with respect to).

VARIABILITY

List Name	Combination	Attribute	Type	Description
CovarMatrix	type (M), parameter (M), value (M)	type	BE:vartype	The type of variability parameter
		parameter	vector:Real	The random variables for which the covariance is defined. The vector defines the columns of the covariance matrix.
		value	vector:Real	The initial estimate values for the covariance. This is the lower triangular matrix of the covariance matrix and excludes its diagonal.
CorrMatrix	type (M), parameter (M), value (M)	type	BE:vartype	The type of variability parameter
		parameter	vector:Real	The random variables which are correlated. The vector defines the columns of the correlation matrix.
		value	vector:Real	The initial estimate values for the correlation matrix. This is the lower triangular matrix of the correlation matrix and excludes its diagonal.

SDEstimate	type (M), value (M), lo (O), hi (O), fix (O)	type	BE:vartype	The type of variability parameter
		value	Real	The initial estimate for this parameter.
		lo	Real	The lower limit of the estmate. Default is no limit.
		hi	Real	The upper limit of the estimate. Default is no limit.
		fix	Boolean	The parameter value is fixed. Default is false.
VarEstimate	type (M), value (M), lo (O), hi (O), fix (O)	type	BE:vartype	The type of variability parameter
		value	Real	The initial estimate for this parameter.
		lo	Real	The lower limit of the estmate. Default is no limit.
		hi	Real	The upper limit of the estimate. Default is no limit.
		fix	Boolean	The parameter value is fixed. Default is false.

STRUCTURAL

List Name	Combination	Attribute	Type	Description
SructuralEstimate	value (M), lo (O), hi (O), fix (O)	value	Real	The initial estimate for this parameter.
		lo	Real	The lower limit of the estmate. Default is no limit.

		hi	Real	The upper limit of the estimate. Default is no limit.

OBJECTS

List Name	Combination	Attribute	Type	Description
MdlObjInMog	type (M)	type	BE:objType	The type of object.

OBSERVATION

List Name	Combination	Attribute	Type	Description
CountObs	type (M), distn (T)	type	BE:obstype	The type of observation.
		distn	PMF	The distribution use to model this observation.
TteObs	type (M), hazard (M)	type	BE:obstype	The type of observation.
		hazard	ref:Real	The hazard function.

7.3.6 Properties

Owning block	Name	Type	Description
ESTIMATE	algo	BE:estAlgo	The estimation algorithm to use.
SIMULATE	solver	BE:solver	The type of solver to use for simulation.

7.3.7 Builtin Enumerations

Name	Permitted Values
divUse	addl, amt, catCov, cmt, cov, dv, dvid, id, idv, ii, mdv, rate, ss, varLevel
ddvUse	doseTime
varLvlType	parameter, observation
compType	depot, compartment, elimination, transfer, direct, distribution, effect
vartype	cov, corr, sd, var
obstype	count, tte
objType	mdlObj, dataObj, parObj, taskObj
transType	none, ln, logit, probit

estAlgo	saem, foce, fo, focei
solver	stiff, nonStiff

7.4 Modularity and Reuse in MDL

The different objects in MDL are reusable modules that can be put together in different combinations. That means each object must be self contained and does not rely on another object for its correct definition. This is discussed above in Section 1.2.2 on scoping.

The model in MDL is only fully defined when objects are assembled into a Modelling Object Group (MOG). Currently a MOG is assembled from the following object types:

- Data object
- Parameter object
- Model object
- Task object

When assembling the MOG parameters and variables in one object may be related to equivalent symbols in another object. For example, a parameter in a model object may be assigned a value defined in the parameter object. The rules for relating symbols in different objects in the MOG are very simple and is based on name alone. For example, a WT column in a data object is mapped to a WT covariant in a model object. This type of mapping is called "magic mapping" and with one or two exceptions (see below) is the basic type of mapping for symbols between objects in a MOG. Obviously even if all 4 objects that make up a MOG are valid in themselves they may not combine to define a valid model. Therefore, MOG assembly has a number of validation rules to ensure that matched symbols are compatible with each other and that all parameters and variables in the model are initialised. These rules are summarising below:

7.4.1 Data mapping

- The column designated the individual identifier (use is id) is matched using magic mapping to the variability level definition in the model with the same name.
- The independent variable in the model object is matched to the column designated in the individual variable in the data object. They need not have the same name as their equivalence can be inferred.
- The column designated the observation in the data object (use is dv) must have the same name as an observation variability level in the model object.
- All variability levels in the model must be matched to a column in the data object with a use of id, dv or varLevel.
- Symbols defined in the DECLARED_VARIABLES block of the data object may be matched to any variable defined in the MODEL_PREDICTION or OBSERVATIONS block.
- Covariates that are not initialised in the model object must be matched to a covarate column definition in the data object. Magic mapping is used.
- If a covariate is assigned a value or if the covariate represents a transformation of another covariate then any matching definition in the data object is ignored.
- Covariates can only be mapped if they of the same type, i.e. continuous to continuous and categorical to categorical.

- For categorical covariates to match both must contain exactly the same set of category values.
- Covariates can only be matched to columns in the data object with a designation of use is covariate or use is catCov.
- Column names in the data object cannot be matched to variables or parameters in the model object.
- Variables referred to by the dosing column (use is amt) must map to variables in the MODEL_PREDICTION block of the model object.

7.4.2 Parameter Initialisation

The parameters in the model object are either initialised in the model object itself or initialised from the parameter object – according to the following rules:

- If a parameter is initialised in the model object then an equivalent assignment in another object is ignored.
- Parameters are matched using magic mapping.
- A parameter defined in the STRUCTURAL_PARAMETERS block can only be matched to a parameter in the STRUCTURAL block.
- A parameter defined in the VARIABILITY_PARAMETERS block can only be matched to a parameter in the VARIABILITY block.
- Variability parameters in the parameter object with types corr and cov cannot be matched to a parameter in the model object.
- The random variables used in the parameter object to specify correlation or covariance must match to random variables defined in the model object.
- When used in a single correlation or covariance definition all the random variables referred to must share the same variability level on the model object.

7.4.3 Task mapping

- If the task specified is an estimation, then an observation column (use is dv) must be defined in the data object.
- If the task specified is a simulation, then the observation column is ignored (if present).

8 Description of UseCases models in MDL

A set of UseCases (UC) has been prepared to illustrate how different modelling features can be implemented in the Model Definition Language or MDL. These UseCases are located in the "models" folder under the UseCasesDemo project pre-configured in the IDE. The key characteristics represented in each UC can be found in Table I. Please note that development is still ongoing. Some models are not expected to be fully interoperable (estimation is only possible with NONMEM). Some other Use Cases are not included in this release, but are presented here to give an overview of models which will become available in later releases. In addition, a more detailed description of the working UCs is available on the subsequent pages.

Table I: Brief description of UseCases

ID	Dataset	Description	Interoperable	Included in this release
UC1	warfarin_conc.csv	PK model, ODE, single oral administration	YES	YES
UC2	warfarin_conc.csv	PK model, analytical solution	YES	YES
UC3	warfarin_conc_pca.csv	PK and PD outcomes, use of DVID	YES	YES
UC4	warfarin_infusion_oral.csv	Different dosing routes with ODE, use of CMT	YES	YES
UC4_1	warfarin_infusion_oral.csv	Different dosing routes with COMPARTMENTS	YES	YES
UC5	warfarin_conc_sexf.csv	Categorical covariate and covariate transformations	NO	YES
UC6	warfarin_conc.csv	PK model, correlation between random effects	YES	YES
UC7	warfarin_conc_cmt.csv	PK model (1CMT) with COMPARTMENTS	YES	YES
UC8	warfarin_conc_bov_P4 _sort.csv	PK model, between occasion variability	NO	YES
UC9	warfarin_infusion.csv	PK model, IV infusion	YES	YES
UC10	warfarin_conc_cmt.csv	PK model (2CMT) using COMPARTMENTS	NO	NO
UC10_1	warfarin_conc_cmt.csv	As UC10 with different parameterisation	NO	NO
UC11	count.csv	Poisson count data	YES	YES

UC12	binary.csv	Binary outcome data, Bernoulli	NO	NO
UC12_1	binary.csv	Binary outcome data, Binomial	NO	NO
UC13	category.csv	Categorical outcome data	NO	NO
UC14	warfarin_TTE_exact.csv	Time to event data, right censoring and exact	YES	YES
UC14_1	warfarin_TTE_ intervalCensored.csv	Time to event data, interval censored	NO	NO
UC14_2	warfarin_RTTE_ intervalCensored.csv	Repeated time to event data	NO	NO
UC15	warfarin_conc_cmt.csv	Complex PK model using COMPARTMENT, multiple dosing routes	NO	NO
UC16	BIOMARKER_simDATA.csv	Multiple observations, log-transformed outcomes, ODE	NO	NO
UC17	warfarin_conc_SS.csv	Steady state dosing using SS	NO	YES
UC17_1	warfarin_conc_SS.csv	Steady state dosing using SS, II, ADDL	NO	YES

8.1 UseCase1

Warfarin population pharmacokinetic model using ordinary differential equations (ODEs)

Dosing regimen: single oral administration

Dataset:

- ID : Patient identifier (n=32)
- TIME [h]
- WT : Patient's weight [kg]
- AMT : Total drug administered [mg]
- DVID : dependent variable identifier (0: dose,1:PK measurement)
- DV : Warfarin concentration [mg/L]
- MDV : missing dependent variable (0: observation, 1: dosing record)
- logtKG : log transformed patient's body weight standardised to 70 kg

Structural model:

- 1 compartment model using ODE (V, CL, k_a, TLAG)
- 1st order absorption process with lag time
- 1st order elimination process

Covariate model:

- WT on CL and V following allometric principles

$$\text{POP\_CL} \times \left(\frac{WT}{70}\right)^{0.75} \qquad \text{POP\_V} \times \left(\frac{WT}{70}\right)^{1}$$

Variability model:

Inter-individual variability:

- Exponential model for V, CL, k_a and TLAG (this last fix to 0.1) expressed as standard deviation

$$\theta_i = \theta \times e\exp\left(\eta_{\theta_i}\right) \qquad \eta_{\theta_i} \sim \mathcal{N}(0, \omega_\theta)$$

- Correlation between CL and V random variables expressed in correlation scale

Residual error model:

- Combined error model

$$y_{i,j} = IPRED + g \times \varepsilon_{i,j} \qquad \varepsilon_{i,j} \sim \mathcal{N}(0,1) \quad var\left(y_{i,j}\right) = g^2$$

$$g = RUV\_ADD + RUV\_PROP \times IPRED$$

8.2 UseCase2

Warfarin population pharmacokinetic model using analytical solutions

Dosing regimen: single oral administration

Dataset:

- ID : Patient identifier (n=32)
- TIME [h]
- WT : Patient's weight [kg]
- AMT : Total drug administered [mg]
- DVID : dependent variable identifier (0: dose,1:PK measurement)
- DV : Warfarin concentration [mg/L]
- MDV : missing dependent variable (0: observation, 1: dosing record)
- logtKG : log transformed patient's body weight standardised to 70 kg

Structural model:

- 1 compartment model using ODE (V, CL, k_a, TLAG)
- 1st order absorption process with lag time
- 1st order elimination process

Covariate model:

- WT on CL and V following allometric principles

$$\text{POP\_CL} \times \left(\frac{WT}{70}\right)^{0.75} \qquad \text{POP\_V} \times \left(\frac{WT}{70}\right)^{1}$$

Variability model:

Inter-individual variability:

- Exponential model for V, CL, k_a and TLAG expressed as standard deviation

$$\theta_i = \theta \times e\exp\left(\eta_{\theta_i}\right) \quad \eta_{\theta_i} \sim \mathcal{N}(0, \omega_\theta)$$

- Correlation between CL and V random variables expressed in correlation scale

Residual error model:

- Combined error model

$$y_{i,j} = IPRED + g \times \varepsilon_{i,j} \quad \varepsilon_{i,j} \sim \mathcal{N}(0,1) \quad var\left(y_{i,j}\right) = g^2$$

$$g = RUV\_ADD + RUV\_PROP \times IPRED$$

8.3 UseCase3

Population pharmacokinetic and pharmacodynamic model to describe warfarin and PCA response

Dosing regimen: single oral administration

Dataset:

- ID : Patient identifier (n=32)
- TIME [h]
- WT : Patient's weight [kg]
- AGE [years]
- SEX (0: female, 1: male)
- AMT : Total drug administered [mg]
- DVID : dependent variable identifier (0: dose,1:PK measurement, 2: PD measurement)
- DV : Warfarin concentration [mg/L] or PCA measurement
- MDV :missing dependent variable (0: observation, 1: dosing record)
- logtKG ; log transformed patient's body weight standardised to 70 kg

Structural model:

PK model

- 1 compartment model using ODE (V, CL, k_a, TLAG)
- 1st order absorption process with lag time
- 1st order elimination process

PD model

- Indirect response model
- 0 order synthesis (RCPA) and 1st order elimination (KPCA)
- Inhibitory effect of drug concentration on RPCA (synthesis) using an Emax model (EMAX, C50)

Covariate model:

- WT on CL and V following allometric principles

$$\mathrm{POP\_CL} \times \left(\frac{WT}{70}\right)^{0.75} \qquad \mathrm{POP\_V} \times \left(\frac{WT}{70}\right)^{1}$$

Variability model:

Inter-individual variability:

- Exponential model for V, CL, k_a, TLAG (this last fix to 0.1), PCA0, C50 and TEQ (ln(2)/KPCA) expressed as standard deviation

$$\theta_i = \theta \times e\exp(\eta_{\theta_i}) \qquad \eta_{\theta_i} \sim \mathcal{N}(0, \omega_\theta)$$

- Linear model for EMAX

$$\theta_i$$

- Correlation between CL and V random variables expressed in correlation scale

Residual error model:

$$y_{i,j} = IPRED + g \times \varepsilon_{i,j} \qquad \varepsilon_{i,j} \sim \mathcal{N}(0,1) \qquad var(y_{i,j}) = g^2$$

- Combined error model for warfarin

$$g = RUV\_ADD + RUV\_PROP \times IPRED$$

- Additive error model for PCA

$$g = RUV\_ADD$$

8.4 UseCase4

Warfarin population pharmacokinetic model for multiple dosing via different administration routes

Dosing regimen: intravenous infusion followed by oral administration

Dataset:

- ID : Patient identifier (n=32)
- TIME [h]
- WT : Patient's weight [kg]
- AMT : Total drug administered [mg]
- RATE : infusion rate [mg/h]
- CMT : compartment number (1: absorption compartment, 2: central compartment)
- DV : Warfarin concentration [mg/L]
- logtKG : log transformed patient's body weight standardised to 70 kg
- MDV : missing dependent variable (0: observation, 1: dosing record)

Structural model:

- 1 compartment model using ODE (V, CL, k_a, TLAG, FORAL)
- 1st order absorption process with lag time and bioavailability for oral administration
- 0 order input for intravenous infusion
- 1st order elimination process

Covariate model:

- WT on CL and V following allometric principles

$$\text{POP\_CL} \times \left(\frac{WT}{70}\right)^{0.75} \qquad \text{POP\_V} \times \left(\frac{WT}{70}\right)^{1}$$

Variability model:

Inter-individual variability:

- Exponential model for V, CL, k_a and TLAG (this last fix to 0.1) expressed as standard deviation $\theta_i = \theta \times e\exp(\eta_{\theta_i}) \qquad \eta_{\theta_i} \sim \mathcal{N}(0, \omega_\theta)$

- Logit model for FORAL expressed as standard deviation

$$logit(\theta_i) = logit(\theta)$$

- Correlation between CL and V random variables expressed in correlation scale

Residual error model:

- Combined error model

$$y_{i,j} = IPRED + g \times \varepsilon_{i,j} \qquad \varepsilon_{i,j} \sim \mathcal{N}(0,1) \quad var(y_{i,j}) = g^2$$

$$g = RUV\_ADD + RUV\_PROP \times IPRED$$

8.5 UseCase5

Warfarin population pharmacokinetic model incorporating categorical and transformed covariates

Dosing regimen: single oral administration

Dataset:

- ID : Patient identifier (n=32)
- TIME [h]
- WT : Patient's weight [kg]
- AGE [years]
- SEXF (0: male, 1: female)
- AMT : Total drug administered [mg]
- DVID : dependent variable identifier (0: dose,1:PK measurement)
- DV : Warfarin concentration [mg/L]
- MDV :missing dependent variable (0: observation, 1: dosing record)

Structural model:

- 1 compartment model using ODE (V, CL, k_a, TLAG)
- 1st order absorption process with lag time
- 1st order elimination process

Covariate model:

- WT on CL and V following allometric principles
- SEX and AGE on CL with an exponential model

$$\text{POP\_CL} \times \left(\frac{WT}{70}\right)^{0.75} \times e^{\beta_{SEX=1}} \times e^{\beta(AGE-40)} \qquad \text{POP\_V} \times \left(\frac{WT}{70}\right)^{1}$$

Variability model:

Inter-individual variability:

- Exponential model for V, CL, k_a and TLAG (this last fix to 0.1) expressed as standard deviation

$$\theta_i = \theta \times e\exp\left(\eta_{\theta_i}\right) \qquad \eta_{\theta_i} \sim \mathcal{N}(0, \omega_\theta)$$

- Correlation between CL and V random variables expressed in correlation scale

Residual error model:

- Combined error model

$$y_{i,j} = IPRED + g \times \varepsilon_{i,j} \qquad \varepsilon_{i,j} \sim \mathcal{N}(0,1) \quad var\left(y_{i,j}\right) = g^2$$

$$g = RUV\_ADD + RUV\_PROP \times IPRED$$

8.6 UseCase6

Warfarin population pharmacokinetic model

Dosing regimen: single oral administration

Dataset:

- ID : Patient identifier (n=32)
- TIME [h]
- WT : Patient's weight [kg]
- AMT : Total drug administered [mg]
- DVID : dependent variable identifier (0: dose,1:PK measurement)
- DV : Warfarin concentration [mg/L]
- MDV : missing dependent variable (0: observation, 1: dosing record)
- logtKG : log transformed patient's body weight standardised to 70 kg

Structural model:

- 1 compartment model using ODE (V, CL, k_a, TLAG)
- 1st order absorption process with lag time
- 1st order elimination process

Covariate model:

- WT on CL and V following allometric principles

$$\text{POP\_CL} \times \left(\frac{WT}{70}\right)^{0.75} \qquad \text{POP\_V} \times \left(\frac{WT}{70}\right)^{1}$$

Variability model:

Inter-individual variability:

- Exponential model for V, CL, k_a and TLAG (this last fix to 0.1) expressed as standard deviation

$$\theta_i = \theta \times e\exp\left(\eta_{\theta_i}\right) \qquad \eta_{\theta_i} \sim \mathcal{N}(0, \omega_\theta)$$

- Correlation between CL, V and k_a random variables expressed in correlation scale

Residual error model:

- Combined error model

$$y_{i,j} = IPRED + g \times \varepsilon_{i,j} \qquad \varepsilon_{i,j} \sim \mathcal{N}(0,1) \quad var\left(y_{i,j}\right) = g^2$$

$$g = RUV\_ADD + RUV\_PROP \times IPRED$$

8.7 UseCase7

Warfarin population pharmacokinetic model using Comparments

Dosing regimen: single oral administration

Dataset:

- ID : Patient identifier (n=32)
- TIME [h]
- WT : Patient's weight [kg]
- AMT : Total drug administered [mg]
- CMT : compartment number (1: absorption compartment)
- DVID : dependent variable identifier (0: dose,1:PK measurement)
- DV : Warfarin concentration [mg/L]
- MDV : missing dependent variable (0: observation, 1: dosing record)
- logtKG : log transformed patient's body weight standardised to 70 kg

Structural model:

- 1 compartment model using ODE (V, CL, ka, TLAG, FORAL)
- 1st order absorption process with lag time and bioavailability (fix to 1)
- 1st order elimination process

Covariate model:

- WT on CL and V following allometric principles

$$\text{POP\_CL} \times \left(\frac{WT}{70}\right)^{0.75} \qquad \text{POP\_V} \times \left(\frac{WT}{70}\right)^{1}$$

Variability model:

Inter-individual variability:

- Exponential model for V, CL, k_a and TLAG (this last fix to 0.1) expressed as standard deviation

$$\theta_i = \theta \times e\exp\left(\eta_{\theta_i}\right) \qquad \eta_{\theta_i} \sim \mathcal{N}(0, \omega_\theta)$$

- Correlation between CL and V random variables expressed in correlation scale

Residual error model:

- Combined error model

$$y_{i,j} = IPRED + g \times \varepsilon_{i,j} \qquad \varepsilon_{i,j} \sim \mathcal{N}(0,1) \quad var\left(y_{i,j}\right) = g^2$$

$$g = RUV\_ADD + RUV\_PROP \times IPRED$$

8.8 UseCase8

Warfarin population pharmacokinetic model

Dosing regimen: single oral administration

Dataset:

- ID : Patient identifier (n=32)
- TIME [h]
- WT : Patient's weight [kg]
- AGE [years]
- SEX (0: female, 1: male)
- AMT : Total drug administered [mg]
- OCC : occasion identifier (1: 1st occasion , 2: 2nd occasion)
- MDV : missing dependent variable (0: observation, 1: dosing record)

Structural model:

- 1 compartment model using ODE (V, CL, k_a, TLAG)
- 1st order absorption process with lag time
- 1st order elimination process

Covariate model:

- WT on CL and V following allometric principles

$$\text{POP\_CL} \times \left(\frac{WT}{70}\right)^{0.75} \qquad \text{POP\_V} \times \left(\frac{WT}{70}\right)^{1}$$

Variability model:

Inter-individual variability:

- Exponential model for V, CL, k_a and TLAG (this last fix to 0.1) expressed as standard deviation

$$\theta_i = \theta \times e\exp\left(\eta_{\theta_i}\right) \qquad \eta_{\theta_i} \sim \mathcal{N}(0, \omega_\theta)$$

Between occasion variability:

- Between occasion variability on V and CL
- Correlation between CL and V between occasion random variables expressed in correlation scale

Residual error model:

- Combined error model

$$y_{i,j} = IPRED + g \times \varepsilon_{i,j} \qquad \varepsilon_{i,j} \sim \mathcal{N}(0,1) \quad var\left(y_{i,j}\right) = g^2$$

$$g = RUV\_ADD + RUV\_PROP \times IPRED$$

8.9 UseCase9

Warfarin population pharmacokinetic model

Dosing regimen: intravenous infusion

Dataset:

- ID : Patient identifier (n=32)
- TIME [h]
- WT : Patient's weight [kg]
- AMT : Total drug administered [mg]
- RATE : infusion rate [mg/h]
- DV : Warfarin concentration [mg/L]
- MDV : missing dependent variable (0: observation, 1: dosing record)
- logtKG : log transformed patient's body weight standardised to 70 kg

Structural model:

- 1 compartment model using ODE (V, CL)
- 0 order input
- 1st order elimination process

Covariate model:

- WT on CL and V following allometric principles

$$\text{POP\_CL} \times \left(\frac{WT}{70}\right)^{0.75} \qquad \text{POP\_V} \times \left(\frac{WT}{70}\right)^{1}$$

Variability model:

Inter-individual variability:

- Exponential model for V and CL expressed as standard deviation

$$\theta_i = \theta \times e\exp\left(\eta_{\theta_i}\right) \qquad \eta_{\theta_i} \sim \mathcal{N}(0, \omega_\theta)$$

- Correlation between CL and V random variables expressed in correlation scale

Residual error model:

- Combined error model

$$y_{i,j} = IPRED + g \times \varepsilon_{i,j} \qquad \varepsilon_{i,j} \sim \mathcal{N}(0,1) \quad var\left(y_{i,j}\right) = g^2$$

$$g = RUV\_ADD + RUV\_PROP \times IPRED$$

8.10 UseCase11

Poisson count data model

Dosing regimen: NA

Dataset:

- ID : Patient identifier (n=100)
- TIME [h]
- CP : Drug concentration acting as covariate [mg/L]
- DV : Number of counts
- MDV : missing dependent variable (0: observation, 1: dosing record)

Statistical model:

- Poisson distribution model

Covariate model:

- Linear effect of drug concentration on baseline count parameter on the logarithmic domain to ensure parameter positivity

Variability model:

Inter-individual variability:

- Exponential model for baseline count parameter expressed as variance

$$\theta_i = \theta \times e\exp\left(\eta_{\theta_i}\right) \qquad \eta_{\theta_i} \sim \mathcal{N}(0, \omega_\theta)$$

8.11 UseCase14

Time to event model for exact and right censored information

Dosing regimen: NA

Dataset:

- ID : Patient identifier (n=32)
- TIME [h]
- TRT : Treatment identifier
- DV : Event identifier (0: no event or right censored, 1: event at exact time)

Statistical model:

- Constant hazard model

Covariate model:

- Proportional covariate model of treatment on the baseline hazard

8.12 UseCase17

Warfarin population pharmacokinetic model at steady-state

Dosing regimen: single oral administration

Dataset:

- ID : Patient identifier (n=32)
- TIME [h]
- WT : Patient's weight [kg]
- AGE [years]
- SEX (0: female, 1: male)
- AMT : Total drug administered [mg]
- SS : Steady-state
- II : Dosing interval
- DVID : Dependent variable identifier (0: dose,1:PK measurement)
- DV : Warfarin concentration [mg/L]
- MDV : Missing dependent variable (0: observation, 1: dosing record)
- logtKG : Log transformed patient's body weight standardised to 70 kg

Structural model:

- 1 compartment model using ODE (V, CL, k_a)
- 1st order absorption process
- 1st order elimination process

Covariate model:

- WT on CL and V following allometric principles

$$\text{POP\_CL} \times \left(\frac{WT}{70}\right)^{0.75} \qquad \text{POP\_V} \times \left(\frac{WT}{70}\right)^{1}$$

Variability model:

Inter-individual variability:

- Exponential model for V, CL and k_a expressed as standard deviation

$$\theta_i = \theta \times e\exp\left(\eta_{\theta_i}\right) \qquad \eta_{\theta_i} \sim \mathcal{N}(0, \omega_\theta)$$

- Correlation between CL and V random variables expressed in correlation scale

Residual error model:

- Combined error model

$$y_{i,j} = IPRED + g \times \varepsilon_{i,j} \qquad \varepsilon_{i,j} \sim \mathcal{N}(0,1) \quad var\left(y_{i,j}\right) = g^2$$

$$g = RUV\_ADD + RUV\_PROP \times IPRED$$

9 Implementing M&S workflows using the ddmore R package

Along with the collection of MDL models, a set of R scripts have been prepared to illustrate how these models can be used in a M&S workflow using the "ddmore" R package:

- Initialisation of the R console
- Read and parse MDL files in R
- Exploratory graphical analysis of the data
- Parameter estimation with Monolix
- Evaluation of the model executed in Monolix using Xpose
- Parameter estimation of the same model with NONMEM
- Evaluation of the model executed in NONMEM using Xpose
- Change estimation method to FOCEI and re-estimation of the model via PsN
- Bootstrap in Perl speaks NONMEM to evaluate parameter precision
- Updating parameter estimates in the MDL Parameter Object using MLE values from NONMEM estimation
- Performing a Visual Predictive Check (VPC) in PsN using MLE values from NONMEM
- Simulate new observed values using the simulx function in the mlxR package.

All the R scripts have been commented to guide the tester though the code and provide information regarding the new 'ddmore' functions. Additional information of the functions can be obtained from the R help typing the name of the function after "?" (e.g. ? as.PharmML) or directly navigating thorough the help files.

Please note that there might be some cases were not all the steps can be performed, these information is also provided in the R script. A link to a set of html reports against which you could confront your results will be provided shortly after the release at the DDMoRe Forum (http://www.ddmore.eu/forum).

9.1 Execution of the R script

There are different ways of executing the R scripts. We recommend to run the scripts line by line so the user gets familiarised with the different functions and they output produced. However, and given the execution of some tasks might take several minutes and in some cases up to an hour, two alternative mechanism are possible: (i) the "source" option and (ii) the "spin" function of the knitr R package.

To run the R script line by line:

Navigate to the ./scripts subfolder

Open the relevant file in the MDL-IDE editor by double-clicking. Select any code lines you wish to execute by marking them with your cursor, and press CTRL+R+R to execute them. You can also modify the code to explore different options.

The tasks has ended once the cursor in the console appears highlighted in blue again

When using this option, and depending upon the command executed, the results will be returned to your workspace (e.g. folder containing the results of a estimation in Monolix , generation of pdf files), to your console (e.g. information of the estimation process, evaluation of the results) or to the R graphics window (e.g. plot of data).

To run the R script via source:

Navigate to the ./scripts subfolder

Right click on the R file named you wish to execute. Then select --> “Run as” --> “R script in R submitting directly”.

The task has ended once the cursor in the console appears highlighted in blue again

This option will also return information as if the code would have been run line by line.

To run the R script via spin function:

load the knitr library, e.g. `library(knitr)`

Spin the file using the full path to the R script, e.g.
`spin(file.path(Sys.getenv("MDLIDE_WORKSPACE_HOME"),"UseCasesDemo","scripts","UC2_Prod4.1_Beta.R"))`

The task has ended once the cursor in the console appears highlighted in blue again

This option will return the results to your workspace, and will also generate an html report collecting all the commands and their respective output.

If you have any question or experience any problem while executing the R scripts, do not hesitate to contact us via the DDMoRe Forum (http://www.ddmore.eu/forum).

10 LIST OF KNOWN ISSUES

The following issues are known to the MDL Developers at the time of the Public Beta release of the MDL and the Interoperability Framework Standalone Execution Environment. (11th December 2015). Many of these have been noted in the description of the MDL Objects in the previous sections. They are collated here for reference.

10.1 Data Object

10.1.1 DATA_INPUT_VARIABLES

- If time is used as idv, the variable name used in the MDL needs to be TIME → Otherwise the SO is not correctly populated for Monolix
- For Monolix, If covariates are to be used in the model, the name given to the variable in MDL needs to match the name in the dataset header , taking also into account lower or upper cases → Monolix uses the header name to retrieve the covariate information
- Standard NONMEM ignore statements are not supported in the current version. Any data processing needs to be done before using the dataset for a task in Monolix or NONMEM. Special treatment of the header row for NONMEM is not needed since a logic to ignore that row is automatically implemented
- In NONMEM, models with dosing to multiple compartment need a CMT column in the dataset indicating to which compartment the dose is. This compartment number needs to match the ODE number specified in the DEQ section. Monolix does not support dosing to multiple compartments unless COMPARTMENTS (PK macros are used)

10.1.2 SOURCE

- MDL file and data need to be collocated for Monolix execution

10.2 Parameter Object

10.2.1 STRUCTURAL

- Structural parameters without IIV associated need to be redefined in the INDIVIDUAL_PARAMETER block, otherwise they are not correctly translated to MLXTRAN (see INDIVIDUAL_PARAMETERS)
- Lag time parameter does not have a key word that can be recognized by the target tools, unless COMPARTMENTS (PK macros) are used to define the structural model. This means that lag time mechanism has to be explicitly defined in the model (see UseCase1 for an example) and estimation of IIV is not supported when using FOCE or FOCEI in NONMEM
- Bioavailability parameter does not have a key word that can be recognized by the target tools, unless COMPARTMENTS (PK macros) are used to define the model. This means that bioavailability has to be implemented in the model equations (see UseCase4). It also implies that bioavailability cannot be used to define initial amount in a compartment different than the one specified under the AMT column (common practice in previous versions of NONMEM to initialise compartments)

10.2.2 VARIABILITY

- Correlation parameter only works in Monolix and Simulx if correlation is specified, not covariance → as a temporary solution, always use correlation scale
- When Simulx is used in a model with correlation between parameters, the default Monolix name for correlation is used (e.g. correlation between V and CL, r_V_CL) → this is triggered by the lack of name in MDL for the correlation parameters. To solve this the parameter name in the vector of parameters used as input in Simulx needs to map the correlation name given by Simulx.
- The specified scale (var or sd) needs to map the one indicated under the modelObj. The framework does not perform any transformation of scales and the one indicated in the modelObj is the one used in the executable model.
- The name used for the 'use is varlevel' variable in cannot contain an underscore when using NONMEM as target.

10.3 Model Object

10.3.1 COVARIATES

- Monolix makes a difference between covariates which act at the parameter level (only time invariant), and covariates that modified other variables in the model, so called regressors. MDL does not make an explicit differentiation, but covariates used to define type 3 individual parameters, or to define derived covariates that are used afterwards in this type of parameter declaration, will be translated as regressors. For the COVARIATES block to be Monolix-compatible, any covariate has to be either:
    - Used in a type 3 definition as a covariate, but not elsewhere (e.g. in MODEL_PREDICTION)
    - Used in MODEL_PREDICTION, but not elsewhere (e.g. in a type 3 definition)
    - Used in an equation producing a transformed covariate, but not elsewhere. It is not possible to have a chain of redefined covariates, or a redefined covariate depending on several other covariates.
- Regressors are currently not supported in Simulx
- Categorical covariates are not supported by Monolix
- Categorical covariates cannot be used in conditional statements → a walkaroun is to use the covariate as a categorical covariate instead.

10.3.2 GROUP_VARIABLES

- Any statement placed into GROUP_VARIABLES will make the model non Monolix compatible:
    - Monolix only supports linear / “linear after transformation” relation between time invariant covariates, structural parameters and random effects (type 3 definition of individual parameters). No other transformation of the structural parameters will be compatible with Monolix.
    - Monolix does not support the declaration of constants in this block (they have to be placed under MODEL_PREDICTION). Otherwise they will also be interpreted as parameter in the longitudinal section (ticket 359)
    - Transformation of covariates need to be placed under COVARIATES block

10.3.3 INDIVIDUAL_PARAMETERS

- To be Monolix compatible, INDIVIDUAL_VARIABLES has to contain exclusively two types of statements:
    - type 3 declaration of individual parameters
    - Simple assignment to define parameters without IIV associated (e.g. BASE=POP_BASE). Non-IIV parameters defined through simple assignments are supposed normally distributed, i.e. they can become negative. If positivity is desired, a type 3 definition has to be used fixing IIV to 0.
- For NONMEM support, individual parameters need to be defined in the same order as the structural and variability parameters appear in the STRUCTURAL block due to a dependency issue

10.3.4 MODEL_PREDICTION

- For the moment, variables need to be used sequentially and after they have been declared
- For Monolix compatibility, any variable used in MODEL_PREDICTION has to be either:
    - the independent variable
    - defined in MODEL_PREDICTION
    - declared in INDIVIDUAL_VARIABLES in the way stated above
    - defined as a covariate in MDL, with the restriction mentioned in the covariate block (will be a regressor in PharmML),
    - defined as a data-derived variable (not fully functional yet, see ticket 435).

  This implies in particular that STRUCTURAL_PARAMETERS, VARIABILITY_PARAMETERS, and random variables defined in RANDOM_VARIABLES_DEFINITION cannot be used in MODEL_PREDICTION (an IDE validation could produce an interoperability warning at this level).

10.3.5 OBSERVATION

- Only standard residual error model functions are supported. This means that equation based assignments are not supported in this block.
- For Monolix compatibility, different observation types have to use different STRUCTURAL_PARAMETERS and random variables.
- Nonmem-formatted datasets describing time-to-event are not compatible with Monolix estimation due to the different handling of the record describing entering of the group at-risk. In Nonmem, it’s an MDV==1 entry, whereas in Monolix, it’s an MDV==0 entry.

11 List of supported and unsupported features

Interoperability framework features by release

		Beta release	Future release(s)
High-level overview			
Language standards		MDL v. 7.0	TBD
		PharmML v. 0.6.1	
Key software components		MDL-IDE v.1.3.0	TBD
		Monolix v. 4.3.2	
		NONMEM v. 7.3	
		R v. 3.0.3	
		Xpose v. 4.5.3	
		PsN v. 4.4.8	
		Simulx v. 2.1.1 (mlxR) / 1.1.0 (mlxLibrary)	
Distribution/deployment			
Stand-alone execution environment		x	x
Integration within existing IT infrastructure			x
Features/tasks			
Estimation	Monolix	x	x
	NONMEM	x	x
	winBUGS		x
Graphical diagnostics	R (user-defined)	x	x
	Xpose	x	x
Re-sampling/simulation-based diagnostics	Bootstrap in PsN (+NONMEM)	x	x

		Beta release	Future release(s)
	VPC in PsN (+NONMEM)	x	x
Simulation	MATLAB		x
	SimCyp		x
	Simulx	x	x
Optimal design	PFIM		x
	PopEd		x
Models supported			
Outcome types	Continuous (single or multiple)	x	x
	Binary		x
	Count	x[11]	x[12]
	Categorical		x
	Time-to-event	x[13]	x[14]
Ways to describe the model	ODEs (user-defined)	x[15]	x[5]
	Closed form analytical solutions (user-defined)	x[16]	x[6]
	Short-hand macro notation	x[17]	x[18]
	One-line PK library functions		x[19]

[11] Poisson-type.
[12] Non-Poisson type.
[13] Exact/right-censored type. NOTE: It is currently not possible to combine a PK model directly with a time-to-event PD model. PK concentrations should instead be included as a column in the data set.
[14] Incl. interval-censored type.
[15] Resolves to: ADVAN 13 + $DES in NONMEM; ODEs in Monolix.
[16] Resolves to: $PRED in NONMEM; analytical solution in Monolix.
[17] Resolves to: ADVAN 13 + $DES in NONMEM; PK macros in Monolix.
[18] Resolves to: ADVAN 1-4/10-12 in NONMEM if one of these types, otherwise to ADVAN 13 + $DES; PK macros in Monolix.
[19] Resolves to: ADVAN 1-4/10-12 in NONMEM if one of these types, otherwise to ADVAN 13 + $DES; PK macros in Monolix.

		Beta release	Future release(s)
Administration/dosing/resetting support	Single or multiple dosing schedules	x	x
	Single or multiple administration routes	x	x
	Zero- or first-order input	x	x
	Steady-state dosing (SS, II, ADDL)	x[20]	x[10]
	Non-dosing related compartment resetting		x
Variability model types	Inter-individual variability	x	x
	Inter-occasion variability	x[21]	x
	Higher levels of variability		x
	Correlation between random effects	x[22, 23]	x
	Prior distributions		x
	Mixture models		x
Covariate types	Continuous (constant or time-varying)	x	x
	Categorical	x	x

[20] There is a known bug in the current version of Monolix, which means that ADDL is not handled properly.
[21] Inter-occasion variability is not currently supported in the translation to Monolix.
[22] There is a limitation in the current translation to NONMEM, which means that random effects parameters for inter-individual as well as inter-occasion variability have to be defined in MDL in an order that ensures that the resulting OMEGA matrix in NONMEM is a block-diagonal band matrix.
[23] There is a known bug in the current translation to NONMEM, which means that the OMEGA BLOCK statement for correlated parameters will always appear before the OMEGA statements for un-correlated parameters. To ensure a correct model in NONMEM, the user must make sure to define the correlated random effects parameters before the uncorrelated random effects parameters in MDL.

12 Future MDL plans:

12.1 Introduction

Future versions of MDL will extend currently released functionality, scope of data types, parameter features, model and task types. We will endeavour to address limitations and workarounds in MDL for the Public Beta Release and provide a more full featured language.

Some features described below are already well advanced in definition of MDL but were not able to be implemented fully in time for the Public Beta Release. Others will need more work to fully define and specify how these features will be implemented in MDL.

It is difficult to guarantee which features will be present in future releases due to limitations of time and resource within the project. The following chapter gives the user some idea of the scope of intended features for future release.

12.2 General features

We aim to support a process of annotation of the MDL model which will facilitate upload of annotation information to the DDMoRe model repository.

This will include annotation of units.

12.3 Data Object

We anticipate the future versions of MDL will support:

12.3.1 DATA_INPUT_VARIABLES block

- Extend the support for NONMEM standard features of data specification:
- Reset compartment
- Infusion rate / duration estimation
- Support for EVID
- Improved support for defining time-dependent covariates or regressors.

12.3.2 SOURCE block

- Specification of a “header” row in the dataset providing data column names. This would then mean that the DATA_INPUT_VARIABLES block need not specify data columns in order and that any data column not appearing in DATA_INPUT_VARIABLES would automatically have “use is ignore”.
- Specification of a number of rows to skip before reading the data file
- Additional input data formats (beyond CSV to include e.g. SAS transport files)
- Additional data structures (beyond nonmemFormat)
- Inline data – data defined in column or R named list format.
- Data defined through an R script.

12.4 Parameter Object

We anticipate the future versions of MDL will support:

12.4.1 STRUCTURAL and VARIABILITY block

- Vector and matrix support for STRUCTURAL and VARIABILITY blocks to enable specification of multivariate distributions.
- Specification of a superset of parameters in the Parameter Object which would apply across more than one model, with the Model Object then containing component models which would have their own parameters, for example a PK component model and a PD component model.

12.4.2 VARIABILITY block

- Matrix support within the VARIABILITY block to enable full specification of variance-covariance and standard deviation-correlation matrices.

12.5 Model Object

We anticipate the future versions of MDL will support:

12.5.1 RANDOM_VARIABLE_DEFINITION block

- Allowing the user to define individual parameter distributions (e.g. CL, V, KA), covariates distributions.
- Future versions of MDL will use the Prob-Onto ontology of random variable distributions as defined in PharmML. This will enable a wider range of distributions to be used for between subject variability, residual unexplained variability and prior distributions.

12.5.2 COVARIATES block

- Interpolation in covariates / regressors through use of interpolation functions.
- Improved support for time-varying covariates / regressors to ensure that translation to target software is robust.
- Specification of covariate models including error in measurement of covariates, uncertainty in dose amounts or dose times or observation times.

12.5.3 MODEL_PREDICTION block

- Definition of the structural model using “one line” PK and PD library models.
- Mapping of standard models (specified via COMPARTMENTS and the PK library models described above) to closed-form solutions in target software (where these are available)
- The ability to combine COMPARTMENT and DEQ specification of structural models.
- The ability to specify mixture models.
-

12.5.4 OBSERVATION block

Future versions of MDL will use the Prob-Onto ontology of random variable distributions as defined in PharmML. This will extend the range of different observation distributions which can be encoded in MDL.

12.5.4.1 Binary data

In this version of MDL, binary data (where outcomes are 0, 1 in the observed data) are encoded as categorical outcomes with two categories. Future versions of MDL will be able to use the Prob-Onto definitions of Bernoulli and Binomial distributions.

The syntax for Binary data outcomes is

```
<<OUTCOME VARIABLE NAME>> : {type is discrete
                            withCategories{ <<category1>>, <<category2>> },
                            distn = Bernoulli(category = <<category1 or category2>>,
                                              probability = <<VARIABLE NAME>> )
                            }
```

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

```
INDIVIDUAL_VARIABLES{
   logit(indiv_BASE) = 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

OBSERVATION{
  Y : { type is discrete withCategories{none, event},
        distn = Bernoulli(category = event,
                          probability = P1) }
}
```

Again, 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 to facilitate interoperability across target software. 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. The Bernoulli distribution needs to have the category specified along with the associated probability. Since the model does not have an explicit link to the 0,1 outcomes in the data, we must define explicitly which category is being modelled.

12.5.4.2 Categorical data

In this version of MDL, categorical outcomes are modelled as a categorical distribution only – ordered categorical data or adjacent categories models must have the probability of being in each group calculated explicitly before use in the categorical distribution. Future versions of MDL will expand the definition of outcomes to other categorical outcome types using the Prob-Onto definitions.

The syntax for Categorical data outcomes is:

```
<OUTCOME VARIABLE NAME> : {type is categorical
```

```
                withCategories{ <category1> when <VARIABLE>,
                                <category2> when <VARIABLE>,
                                …
                                <category N> when <VARIABLE>},
        }
```

In the above, "<<category_k>> when <<VARIABLE>>" the <<VARIABLE>> to be used must be on the scale (0,1) and is the probability of <<category_k>>. This <<VARIABLE>> must be defined in the **MODEL_PREDICTION** block.

For example:

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

INDIVIDUAL_VARIABLES{
        indiv_B0 = general(grp=B0, ranEff = [eta_PPV_EVENT])
        indiv_B1 = general(grp=B1, ranEff = [eta_PPV_EVENT])
        indiv_B2 = 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

OBSERVATION{
      Y : {type is categorical
                withCategories{ none when Prob0,
                                mild when Prob1,
                                moderate when Prob2,
                                severe when Prob3}
}
```

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.

12.5.4.3 TIME TO EVENT

Time to event definition in MDL will be extended to cover interval censoring and repeated time to event.

12.6 Task Properties Object

We anticipate the future versions of MDL will support:

- The ability to define target specific settings for algorithms.
- Define a Simulation task
- Define an Optimal design task

12.7 MOG Object

We anticipate the future versions of MDL will support:

- The ability to define variable name mapping between MDL Objects e.g. define how a variable name defined in the Data Object maps to a Model Object variable. This will further enable the modularity and independence of MDL Objects.

12.8 Design Object

The Design object will describe the trial design for use in simulation or design evaluation. It may alternatively include elementary trial designs and constraints for use in finding optimal designs.

12.9 Prior Object

The Prior object will be used to describe prior distributions on model parameters for use in Bayesian tasks such as estimation. It will support parametric, non-parametric and empirical distributions for parameters.

13 Glossary

13.1 Acronyms and Abbreviations

Acronym	Definition
AMT	Dose amount
BOV	Between Occasion Variability (synonym for IOV - Inter-occasion variability)
COV	Covariance
CORR	Correlation
CTS	Clinical Trial Simulation
CWRES	Conditional Weighted Residual
DDMoRe	Drug Disease Model Resources
DoW	Description of Work
DV	Dependent Variable
EFPIA	European Federation of Pharmaceutical Industries and Associations
EMA	European Medicines Agency
EPS	Epsilon - Residual Unexplained Variability random effect
ETA	Empirical Bayes prediction of the inter-individual random effect in a PK or PD parameter
FDA	Food and Drug Administration
FIS	Framework Integration Service
ID	Individual
IDV	Independent variable
II	Inter-dose Interval
IMI	Innovative Medicines Initiative
IMI-JU	Innovative Medicines Initiative Joint Undertaking
IOF	Interoperability framework
IPRED	Individual Prediction
IWRES	Individual Weighted Residual
MDL	Model Description Language

MDL-IDE	Modelling Definition Language Integrated Development Environment
MDV	Missing Dependent Variable
MIF	Mango Integration Framework
MOG	Modelling Object Group
NONMEM	NONLinear Mixed Effects Modelling (see Names)
OBS	Observed (value or data)
PharmML	Pharmacometrics Markup Language
PD	Pharmacodynamic
PK	Pharmacokinetic
PK/PD	Pharmacokinetic - Pharmacodynamic modelling
PRED	Population Prediction
PROV-O	Provenance Ontology
PsN	Perl Speaks NONMEM http://www.uppsala-pharmacometrics.com/software.html
RDF	Resource Description Framework
RUV	Residual Unexplained Variability
sd	Standard Deviation
SE	Standard Error
SEE	Stand-alone Execution Environment
SO	Standard Output
TEL	Task Execution Language. The working name within the DDMoRe project for the ddmore R package used to perform tasks with MDL and define pharmacometric workflow.
TES	Task Execution Server
var	Variance
VPC	Visual Predictive Check
WP	Work Package
WRES	Weighted Residual
XML	Extensible Markup Language

13.2 Definitions and System Names

System Name	Description
Annotation	A description attached to a model or element of a model; see RDF triple.
Bootstrap	Bootstrap is a tool for calculating bias, standard errors and confidence intervals of parameter estimates. It does so by generating a set of new datasets by sampling individuals with replacement from the original dataset, and fitting the model to each new dataset
Connector	A piece of software which enables modelling software to communicate with the interoperability framework
Converter	A piece of software which enables translation across languages (e.g. mdl to pharmML
End-User	A specific user role with privileges to manage only his/her own jobs queues, etc.
Extensible Markup Language (XML)	An open standard for exchanging structured documents and data over the internet that was introduced by the World Wide Web Consortium (W3C).
Metadata	Metadata (metacontent) is defined as data providing information about one or more aspects of the data, such as: ▪ Means of creation of the data ▪ Purpose of the data ▪ Time and date of creation ▪ Creator or author of data ▪ Placement on a computer network where the data was created ▪ Standards used
Monolix	A software for the analysis of nonlinear mixed effects models
MDL-IDE	Graphical user interface of the Interoperability Framework. It provides a framework within which files containing MDL code can be created and edited and Modelling & Simulation workflows can be created and executed.
MDL	MDL is the Model Description Language (formerly MCL - Model Coding Language) the human writable and human readable language designed to describe pharmacometric models.
MLXTRAN	The language used to define models that are executed with Monolix.
NM-TRAN	The language used to define models that are executed with NONMEM.
NONMEM	A software for the analysis of nonlinear mixed effects models. http://www.globomax.com/nonmem.htmhttp://www.globomax.com/nonmem.htm
Ontology	An organization of some knowledge domain that is hierarchical and contains all

	the relevant entities and their relations. An ontology is used to define the relationships and objects that are used to define the RDF Triples that describe the data, models and results with a Pharmacometrics Workflow
R	R is a free software environment for statistical computing and graphics. https://www.r-project.org/
RDF Triple	An RDF Triple is a statement which relates one object to another. It is composed of three parts: - the subject - the entity we are describing - a predication - the relationship - the object - the description DDMoRe uses RDF triples to describe models held within the repository, for example: "MODEL-000034765" "has author" "Lena Friberg"
Task Execution Service (TES)	Performs job-management within the Interoperability Framework.
WinBUGS	Windows implementation of the BUGS (Bayesian Inference Using Gibbs Sampling) project, concerned with flexible software for the Bayesian analysis of complex statistical models using Markov chain Monte Carlo (MCMC) methods. http://www.mrc-bsu.cam.ac.uk/bugs/winbugs/contents.shtmlhttp://www.mrc-bsu.cam.ac.uk/bugs/winbugs/contents.shtml
Pharmacometric Workflow	Tracking the evolution of a model and associated inferences from initial model to final model, capturing metadata and annotations that will facilitate creation of a run record, audit log, QC and reproducibility of all steps within the workflow. Each step in the Pharmacometric workflow may consist of a Task Workflow which defines the procedural steps required to perform a sequence of tasks for a given model.http://www.mrc-bsu.cam.ac.uk/bugs/winbugs/contents.shtml
Task Workflow	A sequence of tasks and procedural steps which can be captured in a scriptable language, like R, facilitating reproducibility of the outputs for a given set of inputs.
Interoperability	One stated of the DDMoRe project is to provide the capability to define the model once and then use it across a variety of target software tools. We call this "Interoperability".
Repository	Another stated aim of the DDMoRe project is to provide a library platform for pre-competitive sharing of models - disease models, drug models etc. We call this library the "Repository".
PharmML	XML based exchange format for encoding of non-linear mixed effect models, trial design and modelling steps used in pharmacometrics. URL: pharmml.org
Standard Output	Tool-independent exchange format intended for storage of results in standardised

(SO)	form, enabling effective data exchange within complex workflows as well as to support the user in assessing, reviewing and reporting a modelling step.