Available online at www.sciencedirect.com

SciVerse ScienceDirect

00 (2015) 000–000

Procedia Computer Science

www.elsevier.com/locate/procedia

The 5th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH)

Implementing a domain-specific language for model-based drug development

Natallia Kokash[a], Stuart L. Moodie[b], Mike K. Smith[c], Nick Holford[d]

[a]*Leiden Institute of Advanced Computer Science, Niels Bohrweg 1, 2333 CA Leiden, The Netherlands*
[b]*Eight Pillars Ltd, Edinburgh, United Kingdom*
[c]*Global Clinical Pharmacology, Pfizer, Sandwich, United Kingdom*
[d]*Department of Pharmaceutical Biosciences, Uppsala University, Sweden*

Abstract

In this paper, we present the implementation of a novel domain-specific language (DSL) for pharmacometric modeling called the Modelling Description Language (MDL). MDL is a modular, declarative language with block structures that allows users to abstract data, processes and mathematical models from auxiliary code, and hence, improves model readability, reusability and opportunities for collaborative research. The main aim of this DSL is interoperability between core software tools used in pharmacometrics. We describe the MDL-IDE, an integrated development environment for MDL, which assists users in writing MDL code. The paper focuses on language constructs and design decisions, briefly explains how models are validated and converted to a machine-readable format for processing by existing model simulation and estimation software tools.

Selection and peer-review under responsibility of Program Chairs.

Keywords: Model-based drug development; pharmacokinetics; domain-specific language; modeling; simulation;

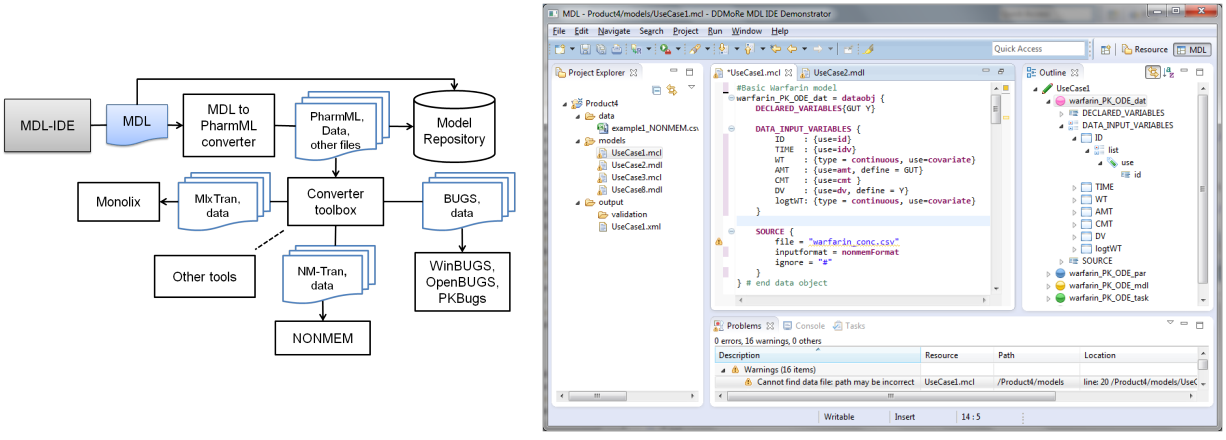
1. Introduction

The use of modeling and simulation in the drug development[1] supports more efficient analysis of preclinical and clinical studies, better decision making and optimized study designs. Specialized software packages (e.g., NONMEM, Monolix, ADAPT II, WinBUGS, Phoenix NLME)[2] and general-purpose statistical and mathematical frameworks (e.g., SAS, S-PLUS, R, Matlab) are commonly used in the pharmacometrics domain.

The proliferation of modeling and simulation applications, incompatibility and proprietary licences of available software, insufficient training, and absence of open resources may cause obstacles for efficient collaboration among researchers[3]. Often a manual translation of the underlying model for each tool is required, not only due to differences in model formulation/language but also due to tool capabilities, software-specific methods and algorithms. Such translation along with the conversion of associated datasets is time-consuming and may introduce errors. It also adds

E-mail address: n.kokash@liacs.leidenuniv.nl

Selection and peer-review under responsibility of Program Chairs.



(a) DDMoRe framework (b) MDL-IDE

Fig. 1. MDL-IDE in the DDMoRe collaboration framework

a burden of validation to confirm that the translation / transcoding was correct. A common specification language and a standard model serialization format would reduce the efforts needed to exchange models.

The purpose of the Drug Disease Model Resources (DDMoRe) project[4] is to provide a framework for collaborative drug discovery and development. Among the main components of this framework are uniform coding standards for model description and exchange. The Modelling Description Language (MDL) is a domain-specific language (DSL) to describe pharmacometric models and tasks associated with these models[5]. The design goal is to be independent of any specific target modeling software. Despite the various software tools describing the same underlying mathematical models, this target independence has proven difficult to achieve in practice. Nevertheless the MDL facilitates understanding of the model by other modelers regardless of their software preferences and previous experience. An important concept in the MDL is the separation of data, parameters, model, task properties and task execution descriptions in terms of independent objects to enable re-usability and interchange of the objects in a sequence of modeling tasks. For example, the user may download an existing model and use it with their own data but choose an alternative method of estimation of model parameters. They can then use the same model object to perform simulation or optimize a future trial design.

In this paper, we describe an implementation of MDL in an Integrated Development Environment (MDL-IDE). We introduce the main language concepts, operators and syntactic constructs, and outline issues related to code validation and type checking. An overview of MDL from the viewpoint of drug development can be found at the project web page (`http://www.ddmore.eu/mdl`).

2. MDL-IDE in the DDMoRe collaboration framework

Figure 1(a) shows basic tool integration within the DDMoRe framework[4]. Users describe models and tasks using the MDL. The MDL can then be automatically translated to an XML-based model exchange format called PharmML[6]. The MDL and PharmML translation may be stored in a shared Model Repository (`http://repository.ddmore.eu`). PharmML code based on the MDL or extracted from the Model Repository may be converted to the input languages of target execution tools. Once the model has been processed by the target tool, the results are returned to the user in a standard output format.

In this framework, the user mainly interacts with the MDL-IDE which provides functionality for the model description and task execution. We use the Eclipse IDE (`http://eclipse.org/`) as a hosting platform for the development of the MDL-IDE. Eclipse also has the benefit of being customizable so that the end product GUI can be tailored to suit the needs of the users in the domain, and can host plug-ins for other useful tools required for the DDMoRe framework.

Implementing a programming language is a complex process that involves:

- writing the parser using a compiler generator
- building the Abstract Syntax Tree (AST)
- implementing access to the AST
- implementing type checking system
- implementing an interpreter for code conversion to a target language.

It is expected nowadays that a programming language comes with an intelligent IDE. The availability of an editor supporting mechanisms such as auto-completion, program outline, automatic error correction, and so on, drastically improves productivity and contributes to the acceptance of a programming language by its target audience.

The Eclipse Xtext framework (`http://www.eclipse.org/Xtext`) provides a powerful and convenient tool for developing DSLs. It generates recurrent artifacts necessary for implementing an IDE within Eclipse. The MDL-IDE, shown in Figure 1(b), is an Eclipse-based editor with a set of windows including project manager, main editing area, outline view, and auxiliary information windows (i.e., tasks, problems, console, etc.). The use of perspectives within Eclipse helps users by presenting task-specific items in a convenient way: arranging windows, displaying useful menu items, and hiding developer functionality that may not be required. The environment allows creation of MDL projects, importing models and data, navigating files in the project, editing MDL files, inserting predefined templates, etc. Validation modules recognize typical user mistakes, i.e., typos in variable names, and perform more advanced checks on the consistency of the model objects. The MDL-IDE release includes the StatET plug-in (`http://www.walware.de/goto/statet`) that enables data and model manipulation using the R language[7]. An R package that provides access to MDL AST for dynamic assembling and execution of models is being developed by the DDMoRe consortium.

3. Syntax overview

An MDL file may include any number of objects in any order. A complete model is defined by a Modeling Object Group (MOG) which includes several objects: (i) a model object, (ii) a parameter object or a prior object, (iii) a data object or a trial design object, and (iv) a task object. An MDL object is defined as follows:

⟨*object*⟩ ::= ⟨*objectName*⟩ '=' '`mdlobj`' '`{`' { ⟨*modelObjectBlock*⟩ } '`}`' | '`parobj`' '`{`' { ⟨*parameterObjectBlock*⟩ } '`}`'
| | '`priorobj`' '`{`' { ⟨*priorObjectBlock*⟩ } '`}`' | '`dataobj`' '`{`' { ⟨*dataObjectBlock*⟩ } '`}`' | '`designobj`' '`{`' { ⟨*designObjectBlock*⟩ }'`}`'
| '`taskobj`' '`{`' { ⟨*taskObjectBlock*⟩ } '`}`' | '`mogobj`' '`{`' { ⟨*mogBlocks*⟩ } '`}`'

We use the Extend Backus Naur Form (EBNF)[8] notation which is more familiar to readers than the xText's LL(*) attribute grammar language[9]. In EBNF, square brackets around an expression indicate that this expression is optional, and curly braces indicate that the expression may be repeated zero or more times.

The content of an MDL object depends on its type and at the upper level is defined by a set of specialized blocks. The blocks for each object type are listed in Table 1. An object can include any number of permitted block types in any order. Blocks do not affect scoping rules and are used to determine the domain-specific semantics of the definitions within them. The content of a block depends on its purpose. Most blocks contain variable declarations. In certain blocks, subblocks can be used to define more specialized variables. For example, a *MODEL_PREDICTION* block contains subblocks *DEQ*, *LIBRARY*, and *COMPARTMENT* to separate differential equations, library calls and compartment models, respectively, from the rest of the code.

Some blocks are used to define properties. For example, *SOURCE* block expects properties that help to define data set file location and type. A property called *file* defines a path to the data file. Additionally, there are blocks with unique content, e.g., the *INLINE* subblock of the *SOURCE* block may be used to specify data in an MDL file directly instead of referring to an external file. A *TARGET_CODE* block provides an environment that allows users to pass verbatim, target specific code (i.e., NM-TRAN, MlXTRAN, R, WinBUGS, Matlab) to any given target and define where it should be placed within the translated code. The use of this block is discouraged to ensure model portability and reusability, the mechanism exists to pass special directives to target software in exceptional cases which cannot be expressed in MDL.

In a nutshell, we can define the content of a typical block as follows:

⟨*block*⟩ ::= ⟨*blockName*⟩ '`{`' { ⟨*variable*⟩ | ⟨*property*⟩ | ⟨*subblock*⟩ } '`}`'

Table 1. MDL objects and blocks

Object	Blocks
Model	IDV, VARIABILITY_LEVELS, COVARIATES, RANDOM_VARIABLES_DEFINITION, INDIVIDUAL_VARIABLES, GROUP_VARIABLES, MODEL_OUTPUT_VARIABLES, MODEL_PREDICTION, STRUCTURAL_PARAMETERS, VARIABILITY_PARAMETERS, OBSERVATION, ESTIMATION, SIMULATION
Parameter	STRUCTURAL, VARIABILITY
Data	DECLARED_VARIABLES, DATA_INPUT_VARIABLES, DATA_DERIVED_VARIABLES, SOURCE
Prior	SOURCE, PRIOR_DISTRIBUTION
Design	COVARIATES, ADMINISTRATION, STUDY_DESIGN, POPULATION_FEATURES, ACTION, SAMPLING, DESIGN_SPACE, HYPER_SPACE
Task	DATA, MODEL, ESTIMATE, SIMULATE, EVALUATE, OPTIMISE
Modeling Object Group	OBJECTS, MAPPING

Variables can be declared without assigned values, initialized with a mathematical expression, a list or a distribution:

⟨*variable*⟩ ::= ⟨*variableName*⟩ ['=' ⟨*expression*⟩ | ':' ⟨*list*⟩ | '∼' ⟨*distribution*⟩]

When initializing a variable with a list, ":" operator is used instead of "=" to emphasize that we do not interpret this statement like a simple assignment: depending on the containing block, a variable described via list attributes may be interpreted in different ways as opposed to a variable assigned an expression which always refers to the expression value. For example, the statement $A : \{\text{deriv} = -ka * A, \text{init} = 0, \text{x0} = 0\}$ in the ***DEQ*** block defines a variable satisfying $\frac{dA}{dt} = -ka * A$, $A(t = 0) = 0$. To the user, the ":" syntax conveys the meaning "has attributes" rather than "is assigned the value".

MDL allows users to define categorical variables, i.e., $SEX : \{\text{type=}categorical(male, female)\}$. Category names must be unique within the scope of the object and can be further used in expressions, i.e., $(SEX == female)$ is a valid boolean expression. This allows the model to utilize boolean expressions without reference to a specific data value (recall that the data and model objects should be independent from each other as much as possible).

Properties are essentially variables with predefined names. A set of valid property names is determined by the containing block type.

MDL mathematical expressions are defined in the usual way and may include function calls, one-dimensional vectors and special values. For example, the expression below

$$OMEGA : \{\text{params} = [ETA\_CL, ETA\_V], \text{value} = [0.01], \text{type} = CORR\}$$

includes vectors and predefined values (i.e., ***CORR***) to define correlation between two parameters.

Vectors can contain mathematical expressions or lists. For example, modelers can assign values to categorical variables with the help of a vector of lists with 2 attributes:

$$\text{define} = [\{\text{category} = male, \text{value} = 0\}, \{\text{category} = female, \text{value} = 1\}].$$

An unusual construct is a piecewise expression used in MDL as an alternative to a conditional (if-then-else) statement:

⟨*piecewiseStatement*⟩ = ⟨*orExpression*⟩ ['when' ⟨*orExpression*⟩ {',' ⟨*orExpression*⟩ 'when' ⟨*orExpression*⟩} ';' | 'otherwise' ⟨*orExpression*⟩ [';']]

The restricted conditional expression is needed to facilitate translation to PharmML that does not support imperative branching statements but allows mathematical piecewise functions.

Lists are flexible constructs inspired by the R language. MDL lists can include any number of named or unnamed attributes:

⟨*list*⟩ ::= '{' ⟨*namedAttribute*⟩ {',' ⟨*namedAttribute*⟩} | ⟨*unnamedAttribute*⟩ {',' ⟨*unnamedAttribute*⟩} '}'

⟨*namedAttribute*⟩ ::= ⟨*attributeName*⟩ '=' ⟨*unnamedAttribute*⟩

⟨*unnamedAttribute*⟩ ::= ⟨*expression*⟩ | ⟨*list*⟩

Recognized attribute names depend on the containing blocks type or, in the case of nested lists, on the outer attribute.

Distributions syntactically resemble function calls with parameters passed either by name or by position:

⟨*distribution*⟩ ::= ⟨*distributionName*⟩ '(' ⟨*namedAttribute*⟩ {',' ⟨*namedAttribute*⟩} | ⟨*unnamedAttribute*⟩ {',' ⟨*unnamedAttribute*⟩} ')'

MDL accepts all distributions recognized by the UncertML (`http://www.uncertml.org/`). In the version 3.0 of UncertML, supported by MDL and PharmML, around 30 distributions are available, i.e., *Bernoulli, BetaDistribution, Binomial, etc.* There are also predefined names for commonly used distribution mixtures (e.g., *DiscreteUnivariateMixtureModel, CategoricalMultivariateMixtureModel*, etc.). Distribution attribute names and expected expression types depend on the distribution type.

MDL predefines numerous domain specific keywords. Many of these keywords originate from NM-TRAN and MlXTRAN, the input languages for NONMEM[10] and Monolix[11] software. Below we list some currently supported terms which are split according to their use. For example, the $\langle useType \rangle$ classifies input variables according to their use in the model, $\langle targetType \rangle$ allows users to specify target software for model processing, $\langle pkMacroType \rangle$ represents body compartments and processes for the compartment-based models, and so on.

⟨*enumType*⟩ ::= ⟨*useType*⟩ | ⟨*targetType*⟩ | ⟨*pkMacroType*⟩ | ...

⟨*useType*⟩ ::= 'id' | 'idv' | 'amt' | 'dv' | 'dvid' | 'ytype' | 'covariate' | 'rate' | 'date' | 'adm' | 'cens' | 'ss' | 'addl' | 'ii' | 'mdv' | 'evid' | 'cmt'

⟨*targetType*⟩ ::= 'NMTRAN_CODE' | 'MLXTRAN_CODE' | 'PML_CODE' | 'BUGS_CODE' | 'R_CODE' | 'MATLAB_CODE'

⟨*pkMacroType*⟩ ::= 'input' | 'direct' | 'effect' | 'compartment' | 'distribution' | 'depot' | 'transfer' | 'elimination''

All MDL objects are independent. Variables declared within an object are not visible from other objects. The exception is the MOG that can access variables of imported objects. An object can be imported to a MOG using the following statement:

⟨*importStatement*⟩ ::= [⟨*varName*⟩ '='] ⟨*objectName*⟩ ['from file' ⟨*URI*⟩]

Here $\langle varName \rangle$ is an optional alias name for an imported object. It is convenient to assign an object reference to an additional variable because objects with the same name can be stored in different files (e.g., different model versions) and the use of aliases makes it easier for users to match variables (i.e., associate data columns or parameter values with model variables). The latter is done in the MOG's *MAPPING* block with the help of assignment statements that involve fully qualified variable names containing an object alias name and a declared variable name from the imported object:

⟨*mappingStatement*⟩ ::= ⟨*fullyQualifiedVarName*⟩ '=' ⟨*fullyQualifiedVarName*⟩

⟨*fullyQualifiedVarName*⟩ ::= ⟨*varName*⟩ '.' ⟨*varName*⟩

Numerous examples of MDL models can be found in DDMoRe repositories, i.e., over 20 open source models are available at `https://sourceforge.net/p/ddmore/use.cases/ci/master/tree/MDL/Product4/`.

4. Validation

A syntactically correct model that complies to MDL grammar can further be validated to ensure its constructs are meaningful, i.e., they can be processed by target execution tools. Initially, each MDL object is validated independently: we check the consistency of declarations within an object. MDL-IDE contains several modules which are responsible for the validation of various MDL concepts, i.e., variables and references, lists, distributions, function calls, properties, and units of measurements. After that, objects assembled to a MOG are validated to comply with each other. Finally, an assembled model is converted to PharmML and the generated code is validated using *libPharmML* library (`https://sites.google.com/site/pharmmltemp/libpharmml`).

Among the basic validation rules for an MDL object is the requirement that each variable is declared once as MDL is a declarative language (although many of the target languages are not). Also, an MDL file must contain MDL objects with unique names. After that we check that all references used in expressions exist, i.e., match variables declared within the current object. In xText, this can be done automatically via cross linking[9], but due to the implicit declaration of variables in MDL and context-dependent interpretation of syntactically similar expressions we had to implement our own reference resolution mechanism.

If an expression includes a function call, the corresponding module checks that the call complies with the function signature. As there are no user defined functions in MDL, signatures of all recognized functions are predefined. A new function can be registered in MDL using a tuple *(name, returned-type, passing-by-name, parameter-sets, returned-variables)*. Here, *returned-variables* may be used to define (global) variables created by the function which can be accessed from the rest of the code.

Among supported functions are standard mathematical functions that expect one or two nameless numeric input parameters and return one numeric value: *abs, exp, factorial, ln, root, min, max*, etc. A *seq* function generates a sequence of numbers given (i) an interval (start and end values) and step size or (ii) a start value, step size and a number of repetitions; it returns a numeric vector and allows parameter passing both by name and by position. Among more advanced functions are commonly used in modeling residual errors: *additiveError, proportionalError, powerError, combinedError*, etc. These functions accept (a subset of) the parameters {*additive, proportional, power, f*} which are references to declared MDL variables.

The MDL to PharmML converter expects a specific set of attributes within a list which it translates to appropriate PharmML constructs. Thus, the validation of lists is context-aware and depends on the containing block. Each attribute is defined as a tuple (*name, expected-types, is-mandatory, default-value*). For each list, the validation method locates its container, defines a set of recognized attributes, and checks that all named attributes in the list belong to this set and their expressions meet type constraints. Each attribute should be defined no more than once, and all mandatory attributes must be specified. For unnamed attributes, the total number of expected attributes and their types are checked. The validation of distribution attributes and properties is analogous.

When a MOG is created, the first task is to validate its general structure, i.e., to ensure that the object group contains exactly one object of each type. This is achieved by locating the definitions of imported objects and deriving their types from the corresponding keywords: *'mdlobj', 'parobj', 'dataobj', 'designobj', or 'taskobj'*. Furthermore, the validation of MOG enforces domain-specific constraints on joint objects. In particular, we check that variables in certain blocks in the model object are among the data input variables defined in the data object. Similarly, the compatibility of model and parameter objects is validated by matching variable sets in *VARIABILITY* and *VARIABILITY_PARAMETERS* and *STRUCTURAL* and *STRUCTURAL_PARAMETERS* blocks. By default, variables with the same names are matched. It is possible to change this mechanism by defining different matching pairs in MOG's *MAPPING* block.

A *units of measurement* defines a physical quantity according to some commonly-accepted convention system. Any other value of the physical quantity can be expressed as a simple multiple of the unit of measurement. Experimental data for analysis mostly refer to some physical quantities and thus these data are measured in certain units. When model, parameter and data objects combined in a MOG have different assumptions about unit measurements, the assembled model may be incorrect. Therefore it is important for MDL to include means for data and model code annotations with units, and for the MDL-IDE to recognize incompatible objects and provide means for their adaptation to ensure unit compatibility.

Users recognize the need for units and unit conversion, but systems with mandatory units on all objects can become too restrictive in practice. Hence, the current implementation provides support for optional unit use. This includes:

- parsing of expressions and validating that the unit measurement is a known metric that makes sense;
- computing values and deriving measurements for mathematical expressions;
- checking unit compatibility for related variables from objects in a MOG.

We allow users to annotate quantities with units of measurement via optional *units* attribute. First, we introduce symbols to represent standard units from relevant unit categories such as *s* (second) for *time*, *L* (liter) for *volume*, *g* (gram) for *mass*, *M* (mole) for *amount of substance*, and *m* (meter) for *length*. Then we allow unit scaling by adding standard prefixes *d* for deci (10^{-1}), *c* for centi (10^{-2}), *m* for milli (10^{-3}), *u* and *mc* for micro (10^{-6}), *n* for nano (10^{-9}), *p* for pico (10^{-12}), and *f* for femto (10^{-15}). For time units, larger metrics can be defined using the following symbols: *min* (minute), *h* (hour), *day*, *week*, and *y* (year).

The basic and scaled units defined above can be combined to valid mathematical expressions using * (multiply), / (divide), and $\hat{}$(power) operators as well as priority brackets and numbers. By parsing such expressions we can identify basic units and associate them with corresponding classes or ontology terms. In MDL-IDE, we use the UOMo[12] Java-based library for unit of measurement manipulation. The UOMo framework provides interfaces and abstract

classes with methods supporting unit compatibility checking, expression of measurement in various units, arithmetic operations on units, classes implementing standard unit types and conversions, parsing and formatting textual representations, and a repository of predefined units. We utilize a standard unit package `eclipse.uomo.units.SI` and define derived units mentioned above via its basic classes GRAM, METRE, CUBIC_METRE, MOLE, and SECOND.

5. Type checking

The more versatile the data types in a language are, the more closely its concepts can match the real world. The initial requirement specification for MDL produced by prospective users did not require explicit data types. In fact, the language specification produced by domain experts was largely example-driven. We realized that many domain-specific constraints and user assumptions cannot be supported without an internal, implicit type system. Among the constraints that affected our approach to the development of the MDL type system are the following:

- All types must be implicitly derived as users are not trained to recognize the importance of types and explicitly declare typed variables.
- Type checking can only be performed statically at the design time as MDL is not an executable language.
- Various restrictions of primitive types (such as Boolean, Integer, Real, String) are required to prevent potential errors in the models and achieve valid conversion to PharmML. In particular, PharmML incorporates UncertML schemas for describing distributions which expect values of subtype ranges such as positive natural or probability (real number from 0 to 1).
- Enumeration types are needed to allow users to classify models and its parts using domain-specific terms.
- Restrictions on MDL mathematical expressions assigned to certain attributes must be supported. For example, we must be able to recognize vectors of probability values, references to derivative variables, etc.

Thus, the first task is to derive types of MDL variables from the context. Below are examples of the rules used to implicitly assign types to declared variables:

- Each declared but not initialized variable is of type *TYPE_REAL*. Each variable has also type *TYPE_REF*.
- A declared variable initialized with a list that contains an attribute type $= categorical$ is of type *TYPE_INT*.
- Each declared object is of type *TYPE_REF_OBJ* and, more specifically, of type *TYPE_REF_OBJ_* $\langle objType \rangle$ where $\langle objType \rangle$ specifies the type of the MDL object.

MDL list attribute, distribution attribute, property and function formal parameter definitions include constraints on their expected types, i.e., an attribute *value* expects a real value (*TYPE_REAL*) while attribute *define* can be assigned a reference (*TYPE_REF*), a list (*TYPE_LIST*), a vector of lists (*TYPE_VECTOR_LIST*), or a piecewise function (*TYPE_PIECEWISE*). MDL-IDE uses a set of derivation rules to determine the actual type of an expression assigned to an attribute, property or function parameter. First, we define types of variables used in an expression with the help of a look-up table. If an expression contains function calls, we obtain their return types from the function descriptors. Given types of all compounds, we can infer the actual type of a mathematical expression. Expected type of an expression assigned to an attribute can be a supertype of an actual type, e.g., we can assign a value of type *TYPE_INT* to an attribute expecting a value of *TYPE_REAL*.

6. Code generation

We need to store models in a shared repository and augment them with additional information such as model version, modification history, semantic annotations (e.g., author, related publications), model simulation and estimation results produced by target tools, and so on. Models from the repository should be easily parsable and convertible to target execution languages. Human-friendly MDL models can be serialized in PharmML[6], a machine-readable XML-based language. The purpose of XML is to enable easy creation of new languages, but its concrete verbose syntax is not adapted to be read by humans. Therefore in the DDMoRe framework we write models in MDL and automatically

convert them to PharmML. The goal of this intermediate language is to make it simpler to convert to many different target tools. Initially, the mapping of MDL to PharmML raised some challenges since the two languages were developed in parallel by separate development teams, but as these languages have evolved over the course of the project the mapping has improved. It remains challenging to integrate the two languages and ensure that concepts in one are consistent with the implementation in the other. The products delivered by the project so far indicate that integration between MDL and PharmML and translation on to target tools is possible and alignment is improving.

Since a reference to a variable in PharmML consists of a variable name and a block name in which this variable was defined, we first collect all declared MDL variables and associate them with PharmML blocks according to their domain-specific use, the combined reference then is used in PharmML expressions. Thus, we retrieve MDL variables for variability, covariate, structural, parameter, and observation models, and create a map that associate them with the corresponding block names.

Basic mathematical expressions in MDL are converted to the corresponding PharmML expressions without significant changes. On the other hand, higher-level constructs such as expressions assigned to attributes and properties, the content of specialized blocks, vectors, and lists require context-dependent interpretation. For example, a categorical variable *SEX* that refers to a data column and defines two categories, *male* and *female*, represented by values 0 and 1 in the data column, respectively:

$$SEX : \{\text{use} = covariate, \text{type} = categorical(male, female), \text{define} = [\{male, 0\}, \{female, 1\}]\}$$

is represented in PharmML as

```
<ColumnMapping>
   <ds:ColumnRef columnIdRef="SEX"/>
   <ct:SymbRef blkIdRef="cm1" symbIdRef="Sex"/>
   <ds:CategoryMapping>
      <ds:Map dataSymbol="0" modelSymbol="male"/>
      <ds:Map dataSymbol="1" modelSymbol="female"/>
   </ds:CategoryMapping>
</ColumnMapping>
```

Note that the expression assigned to the attribute *define* is an array of lists with unnamed attributes. It is not converted to the PharmML's vector but represented in a custom way that better reflects its domain-specific meaning. However, supporting different syntax in MDL for each and every variable use would be impractical, so a generic syntax is adopted to describe similar concepts. Depending on the context, MDL vectors can be translated to PharmML vectors, piecewise functions, sequences of values or special XML tags.

Mathematical function calls are translated to the calls in the XML-based format -since such functions are available in all target environments (with possibly different signatures), it is trivial to translate them from PharmML to these languages. For special functions such as error models, we generate function definitions explicitly. For example, the *combinedError1* function:

$$combinedError1(\text{Real } additive, \text{Real } proportional, \text{Real } f) = additive + proportional * f;$$

is represented as

```
<ct:FunctionDefinition xmlns="http://www.pharmml.org/pharmml/0.6/CommonTypes"
      symbId="combinedError1" symbolType="real">
      <FunctionArgument symbolType="real" symbId="additive"/>
      <FunctionArgument symbolType="real" symbId="proportional"/>
      <FunctionArgument symbolType="real" symbId="f"/>
      <Definition>
            <Equation xmlns="http://www.pharmml.org/pharmml/0.6/Maths">
                  <math:Binop op="plus">
                        <ct:SymbRef symbIdRef="additive"/>
                        <math:Binop op="times">
                              <ct:SymbRef symbIdRef="proportional"/>
                              <ct:SymbRef symbIdRef="f"/>
                        </math:Binop>
                  </math:Binop>
            </Equation>
      </Definition>
</ct:FunctionDefinition>
```

The MDL validation module warns users about possible problems, i.e., unexpected expression types, but it is not always possible to distinguish models that can be translated to valid PharmML from those that must be interpreted in a special way for correct translation. Thus, the translation is feature-based and works for models that adhere to

certain modeling conventions; each new converter release enlarges the set of supported model types (e.g., by adding conversion rules for compartment models, Bayesian models, time-to-event models, etc.).

7. Conclusions and Future Work

In this paper, we presented the implementation of the MDL language. As there is always a certain reluctance to learn a new programming language, especially in a community of non-programmers, many MDL's constructs are adopted from the languages already familiar to the users, such as the R language (e.g., lists), MLXTRAN (e.g., syntax of distributions), and NM-TRAN (e.g., data format and use types). Yet, in contrast to existing languages, MDL focuses on separating models from auxiliary code and achieves reusability by splitting the model into independent objects. The MDL design has evolved alongside the PharmML language as we should be able to convert any valid MDL model to a valid PharmML model.

MDL is a young language which is expected to evolve to support the needs of the growing community. Our first implementation of the MDL-IDE, although somewhat limited in features, was tested in several internal and public DDMoRe training courses with generally positive feedback. We are currently working on the ability to use MDL to code a wider set of features and model types.

Acknowledgements

The research leading to these results received support from the Innovative Medicines Initiative Joint Undertaking under grant agreement 115156, resources of which are composed of financial contributions from the European Union's Seventh Framework Programme (FP7/2007-2013) and EFPIA companies' in kind contribution. The DDMoRe project is also supported by financial contribution from Academic and SME partners.

References

1. Mould, D., Upton, R.. Basic Concepts in Population Modeling, Simulation, and Model-Based Drug Development. *CPT: Pharmacometrics and Systems Pharmacology* 2012;**1**(6).
2. Shargel, L., Yu, A., Wu-Pong, S.. *Applied Biopharmaceutics & Pharmacokinetics*. McGraw Hill Professional; 6th ed.; 2012.
3. Vlasakakis, G., Comets, E., Keunecke, A., Gueorguieva, I., Magni, P., Terranova, N., et al. Landscape on Technical and Conceptual Requirements and Competence Framework in Drug/Disease Modeling and Simulation. *CPT: Pharmacometrics and Systems Pharmacology* 2013;**2**(40).
4. Harnisch, L., Matthews, I., Chard, J., Karlsson, M.. Drug and Disease Model Resources: A Consortium to Create Standards and Tools to Enhance Model-Based Drug Development. *CPT: Pharmacometrics and Systems Pharmacology* 2013;**2**(3):34.
5. Holford, N.H.G., Smith, M.K.. MDL - the DDMoRe Modelling Description Language. Abstracts of the Annual Meeting of the Population Approach Group in Europe (PAGE), www.page-meeting.org/?abstract=2712; 2013.
6. Swat, M.J., et al., . Pharmacometrics Markup Language (PharmML): Opening New Perspectives for Model Exchange in Drug Development. *CPT: Pharmacometrics and Systems Pharmacology* 2015;:316319doi:10.1002/psp4.57.
7. Ihaka, R., Gentleman, R.. R: a Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics* 1996; **5**(3):299–314.
8. Scowen, R.S.. Extended BNF - a Generic Base Standard. Technical report 14977; ISO/IEC; 1998.
9. Xtext 2.6 Documentation. http://www.eclipse.org/Xtext/documentation/2.6.0/Xtext Documentation.pdf; 2014.
10. Bauer, R.J.. NONMEM Users Guide: Introduction to NONMEM 7.3.0. https://nonmem.iconplc.com/nonmem730/nm730.pdf; 2014.
11. Monolix. Version 4.3.2. http://www.lixoft.eu/wp-content/resources/docs/UsersGuide.pdf; 2014.
12. Eclipse UOMo Units of Measurement, Version 0.6. http://www.eclipse.org/uomo/documents/uomo-0.6.pdf; 2013.