Indirect Estimation and Simulation Tool (IEST) Developer Guide

09/Dec/2009

Deanna Isaman
Jacob Barhak, Email: jbarhak@umich.edu
Donghee Lee

University of Michigan
Ann Arbor

Table of Contents

1. Introduction

This document describes the structure of the IEST software for developers. It is intended to help willing developers to continue improving the IEST software. Since the software is released under GPL license, such improvement may be made possible, even after the end of the project and therefore the potential usefulness of this document. This document is not intended for regular users. Users should use the html help system provided with the system.

The document will describe the data structures and algorithms of the system. It will also describe the GUI development environment. Future implementation issues will also be described, especially the estimation engine that was already published previously as a Matlab Prototype under GPL license. This estimation approach is now named “the Lemonade Method”.

Prerequisites to read this manual

- Some understanding of data structures
- Some understanding in regular expressions
- Basic concept about Python and wxPython
- Some understanding about Event driven programming (for GUI)
- Some knowledge of the system itself at the user level

Recommended work environment for development:

- Python 2.5.2 (Essential)
  - The NumPy library (Requires Python), version 1.2.1
  - The SciPy library (Requires Python and NumPy), version 0.7.0
  - WxPython Library (Requires Python), version 2.8.9.1
- WindPdb (for debugging)
- WxGlade (for initial visual arrangement of new GUI forms)
- SPE (Stani’s Python Editor for IDE environment for convenience)
- If running the Matlab prototype, Matlab Version 7.8.0.347 (R2009a) 64-bit (glnxa64) is required – previous versions worked on Matlab R2009a (to support examining the estimation prototype and for other mathematical calculations)

For further information about the project, please visit the project web site:
http://www.med.umich.edu/mdrtc/cores/DiseaseModel/

2. Data Structures

2.1. Entities & Relationships

The system maintains the data structure as depicted in the Entity Relationship Diagram (ERD) below. Data objects are stored in a data file that is actually a zip file bundling several text files. Each text file is a python pickled data entity, and thus can be loaded into python using the pickle module.

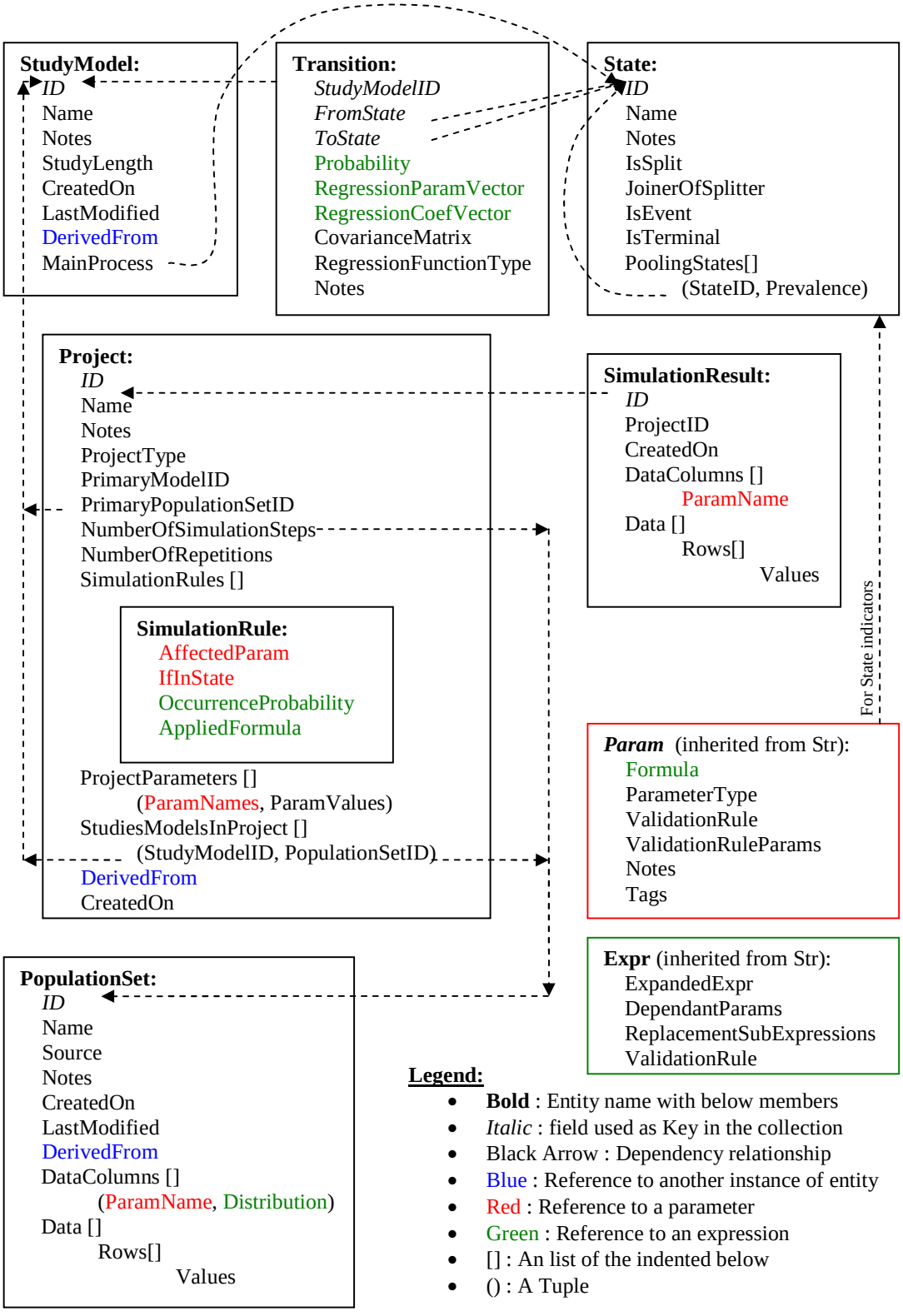
The entities in the system contain references to each other using links. The data structure was initially designed so it can be stored in a relational database. It was later de-regularized to allow convenient storing and handling of the data. Therefore Python list and tuples are used to hold data. Note that many entities in the system reference parameters in the parameter table. These are shown in red in the ERD. Similarly many entities including parameters reference expressions (Expr) and these are marked in green. Note that Expressions are basically classes that are inherited from a string and can reference other parameters. Parameters are a very basic building block of the system and are therefore discussed in much detail below.

The entities in the ERD are defined as classes. Instances (actual objects with data) of this class are held in several collections (lists) the system holds. To simplify thinking, think of entity classes as record structure definition, think of instances of classes as actual records with data in a table, and think of a collection as a table containing the records. Note that collections are basically instances of classes that imitate a list of entities with methods for insertion, deletion, modification, etc. There is only one list for each type of entity held in the data definitions and it is named after the entity it holds with an ‘s’ character at the end, e.g. a Param entity will be hosted in the collection named as Params that is a single instance of ParamsClass. Collections hold a key. In most entities that ID field is used as a key. In Transitions, however, a key is composed of 3 fields and in the case of Params, the key is the parameter name, which is a string.

All the entities in the ERD have Collections/Tables except from the Expr class that is actually used within other entities rather than referenced since it is a string.

In addition to the main entities that are described in the ERD, the system holds the TableClass class that is used to parse tables in compile time and simulation run time.

All these definitions are found in the file DataDef.py that is imported by every other program in the system.



Legend:

- **Bold** : Entity name with below members
- *Italic* : field used as Key in the collection
- Black Arrow : Dependency relationship
- Blue : Reference to another instance of entity
- Red : Reference to a parameter
- Green : Reference to an expression
- [] : An list of the indented below
- () : A Tuple

Entity Relationship Diagram (ERD)

2.2. Data Construction and Validity Checking

Due to the complexities of the data definition, the model data required for simulation and estimation should be constructed in a specific manner. The following section will describe the correct order to construct this data and includes a set of validity checks the system will make in any stage to validate the data integrity:

Order of data construction and appropriate validity checks

1. Define Parameters:
   - Check the validity of the parameter name by using the regular expression.
   - Check the validity of the parameter definition and its rules using the above table.

2. Define States:
   - Simple check of internal data defined properly:
     - Names are valid and do not include parameter state indicator extensions and do not conflict with similar names that will create the same parameter
     - Splitter state is not an event, nor a joiner
     - Check that the sub-process data is valid:
       - The child states were previously defined
       - Does not combine pooled states and sub-process children states
       - Prevalence values are between 0 and 1 and sum to 1 in case of a pooled state.
       - Also check that splitters are defined before joiners and they pair up
       - Make sure that the number of sub-processes is at least the number of splitter states
       - No splitters and joiners or sub-processes are defined for pooled states.
   - When adding to the states collection
     - Check if name conflicts with other states or existing parameters
   - When adding a sub-process state to the states collection – check if all child states exist and no loops are created.

3. Define Study/Model:
   - Check that the main process is a sub-process and not a pooled state
   - Check that a study does not contain splitters, joiners, sub-processes

4. Define Transitions (Splitting transitions should always be defined before joining transitions):
   - Verify that the transition parameters are valid
     - If regression data is defined no probability data is defined and vice versa
     - Regression data is fully defined and is compatible in size and context
   - Check that the states and studies actually exist and are valid and in the states list
   - Check this is not a transition into a sub-process

- If this is a transition for a model, make sure that the transition is not from nor to a NULL state
- A transition from a terminal state is not allowed.
- If transition is from a splitter state
    - Check that the target state has a nesting level one higher then the originating state
    - Check that there are no other transitions into this state from another study (meaning this is the first state in the sub-process)
- If the transition is not to a joiner state and the "from state" is not null or a splitter state, make sure that the states being connected belong to the same father.
- If transition is to a joiner state
    - Check that the target state has a nesting level one lower then the originating state
    - Check that the joiner state is pointing to a valid splitter state that leads to a state in the same sub-process as the state leading to the joiner. Note that this implies that a splitter transition should be defined before the joiner transition.
- To allow adding a transition to a Study/Model, make sure that:
    - The Study/Model is not locked. See definition of locking below.
    - The Study/Model is not used in another project that is not bypassed. See definitions below.

5. Define Population Set:
    - Check that the population is not defined by both data and distribution, i.e. it is not allowed that one or more parameters are defined as distributions while others are defined as data.
    - For data based population, make sure that:
        - The data array conforms in size to the number of columns.
        - Each data item conforms to the validation rule of the parameter. Note that an empty entry is allowed to signify no data. This will bypass any validation rule. Empty data is handled separately during simulation and estimation.
    - For distribution based populations, make sure that:
        - The distributions expressions used are valid. This means either:
            - A valid expression is used that does not use parameters that have empty values that are not other columns
            - Column names used in the expressions do not create an infinite cycle

6. Define a Project:
    - For a simulation project the following validation tests are made:
        - Make sure that the primary model is defined and is a valid model rather than a study
        - Make sure the population set is defined and is a population set based on data rather than on distributions
        - The number of iterations and the number of repetitions is positive.

    - Verify that no estimation based data is defined for the project
    - Make sure the simulation rules are valid and follows these rules:
        - The affected parameter is a valid name and is one of the following types: Coefficient, System Option, Covariate, Intervention, Cost, Quality of Life
        - The IfInState parameter is a valid state indicator
        - The applied formula is not empty
        - If the parameter type is coefficient or system option, then the IfInState and OccurrenceProbability fields are not defined
- For an Estimation project the following validation tests are made:
    - No simulation related data exists
    - Make sure that exactly one model is defined
    - Make sure that at least one study is defined and that no study is repeated twice
    - Make sure population sets defined with the studies are valid
    - Make sure that the estimation parameters are defined properly according to the following rules:
        - The estimation parameters and the values are proper vectors with the same number of members
        - The estimation parameters are either the reserved word 'AllCoefficients' appearing at the beginning of the vector, a parameter that is a coefficient, or a System option. Note that at least one coefficient must be defined.
        - The parameter value is a number and it is valid using the validation rules of the corresponding member in the parameter vector. Where as a number corresponding to AllCoefficients will undergo a validation check with every coefficient used in the model

Data modification rules:
Addition of any entity is allowed if it passes the validation checks above. However, modification and deletion operations require more sophisticated validation to maintain data integrity. A modification operation is composed of deletion operation of the old record followed by an addition operation with the new data. Therefore, the data modification rules explained below referring to a deletion operation, along with the previously described validly checks for an addition operation define the modification rule. Note, however, that if modification fails the system will restore the old record, even if it already was deleted for modification to take place.

To maintain data integrity for the user, there will be several levels of dependency between data entities that will be distinguished:

1. **Locking level:** at this level, data will be locked so that a user will not be able to change it accidentally. For example, after simulation results have been generated, changing project parameters should not be allowed as this may cause the user to assume that results have been generated using other parameters and therefore all entities leading to generation of the simulation results are locked.

2. **Record dependency level:** at this level one record is dependant on another record it references and a change of one record may change/damage/invalidate the data in the other record. For example a parameter is used in a project, changing the parameter should not be allowed as this may render the reference in the project invalid. Note that record dependency can be at some cases bypassed by the system to allow convenient manipulation of the data through the GUI.
3. **Reference level:** At this level one record refers to another record for mostly clerical purposes and this link can be severed without adverse affect on the actual data. For example, when copying an entity, the new record will maintain a link to the old record and severing this link will not change system behavior substantially.

The first two levels cause the system to reject deletion and modification operations in case of locking/dependency. The third level will automatically adjust references within the database at the risk of severing reference links. Below is a formal definition of these concepts:

Locked entities:

- A simulation project is locked if the system holds simulation results for it
- An estimation project is locked if the system has a simulation project created from it, i.e. the estimation project has generated results in the form of a simulation project
- A Study/Model is locked if it is referenced by a locked Project. This means being the primary model in a locked simulation project or a Model/Study in the Model/Study list of a locked estimation project.
- A Population set is locked if it is referenced by a locked Project. This means being the primary population set in a locked simulation project or a population set accompanying a study in the Model/Study list of a locked estimation project.
- A transition is locked if its Study/Model is locked

Dependency Between entities:

The Entity relationship diagram at the beginning of the document represents the main system entities and their dependencies. Each entity is represented as a box with its name in bold at the title and listed below are the data members held in this entity. Arrows represent a “one to many” relationship where the arrow is on the side of the one entity on which many other entities depend on. Member variables colored red are expressions that can contain one or more references to parameters and therefore may depend on parameters. Note that further restrictions can be imposed on these expressions. Also note that there is a special reference between parameters and states, that is valid for state indicators only that depend on a state. Blue fields are not considered a dependency, rather these are considered references rather than dependencies.

A deletion/Modification of any entity that is somehow dependant on the existence of another entity, even through an intermediate expression is not allowed and will be blocked by the system unless a bypass is permitted as defined below.

Bypassing dependency:

Since blocking of dependencies will force the user to construct entities only bottom up, which may be inconvenient, a bypass is designed. With this bypass, it is possible to declare a single project that will be ignored while determining dependencies. Under this bypass, any modification of a Study/Model or population set that are referenced within this project only are allowed. This extends to transitions associated with the study models. If, however, these are used by more than one project, then modification is blocked even if the bypass is defined. Note that bypassing a lock is not allowed. Also note that parameter dependency or state dependency cannot be bypassed as these are basic building stones of the data.

Note that bypassing allows record deletion only in the case of transitions, otherwise bypassing is effective only for modification operations of StudyModels / PopulationSets / Transitions.

Bypassing is triggered by the GUI during drill down operations into the data from the project form. Note that bypassing can create undesirable effects and therefore should be handled delicately by the GUI and some precautions are required.

Reference between entities:

The major entities, Study/Model, Projects, and Population Set include a DerivedFrom field that can hold the ID of another instance of the same entity. This reference is a weak reference maintain by the system for two reasons: 1. Determine locking of estimation projects that their result is a simulation project that references the estimation project that created them. 2. Keep track of which entities where copies from another entity to help the user maintain order and sense in the data.

Upon deletion of the record that was referenced, all referencing Derived from Fields are reset to 0 meaning that the reference is severed. Note, however, that modification of the referenced record will not sever this reference and the reference will be kept. The system performs this automatically to maintain data integrity and avoid broken links.

A summary of the data entry and modification rules:

- Addition of an entity that will not pass the validation rules is forbidden
- Deletion and modification of locked entities is forbidden
- Deletion and modification of entities that have other dependant entities is forbidden unless a modification is allowed by a bypass
- Modification of an entity to an entity that will not pass the validation rules is forbidden
- Reference fields in the same entity will be reset upon deletion of an entity.

2.3. System Parameters

The parameter collection will include the following types of parameters:

1. **Covariate** – provides a name for a column in a result table where information will be stored about a variable. Examples are: Age, Gender, Blood Pressure etc. A covariate can contain an expression interpreting other covariates such as OverWeight defined as BMI>=30.
2. **Intervention** – gives a name for a column in a result table where information will be stored about intervention occurred. For example, if ACE inhibitor was administered, this value will be set to 1.
3. **Cost** – Provides information on costs associated with a specific state. For example the cost of Angina. Cost parameters hold scalar values. However, to calculate these, these can be obtained from a cost wizard or through a mathematical formula.
4. **Quality of Life** – gives a name for a column in a result table where information will be stored about the quality of life. This is similar to cost with a different scale. The CostWizard can also calculate QoL values.
5. **Prevalence** – a probability values that indicates how much a specific state is significant compared to others in the scope of pooled states. For example, how much dialysis and transplant are prevalent in ESRD. The sum of these probabilities should be 1 for each pooled state. This parameter may never be referenced by another parameter and is practically used only to store values rather than to be referenced. Upon use as a state prevalence its value it will be replaced by its value and subsequent change to the parameter will not affect the number used.
6. **Probability** – A parameter that holds a probability. This may be used in the occurrence frequency of a covariate or another parameter during simulation. For example FlipCoin.
7. **Transitions** – the probability to move from one state to another. These can be functions of other parameters. For example, the probability of transition from Normal CVD to Angina. In the case of transition from splitter states, the transition will not mean probability. Rather, the transition probability will be 1 and the transition will mean splitting into parallel sub-processes that will be processed in random order.
8. **Coefficient** – These parameters will be used as unknown parameter values within a transition probability to be calculated during estimation. Coefficient parameters will be assigned with values after the likelihood is optimized.
9. **Function** – gives a name for a function that can be used later during calculations. Each time the function name is used, it will be replaced by the expression it represents. For example, a function that increases the age may be called AgeIncrease and hold the function Age+1.
10. **Table** – A table parameter will hold the entire table definition and values in that table. The table will be defined by unique name just like any other parameter. For example Table1. The table can hold multidimensional data along with dimension names, dimension ranges and table cell values. When a table is accessed during

simulation, then the table will return the value in the table cell corresponding to the columns/rows corresponding to the value currently held by the parameter. Note that since a table is defined by an expression, this is similar to a function parameter. However, there is a stricter validity check with a table parameter that confines the expression to table expressions.

11. **Distribution** – Provide information on the distribution within a population. It is similar to a function and provides the capability of defining marginal distributions for covariates and intervention parameters. Note that the form of the function used as a distribution parameter will be similar to this of a random generator function. The free parameter will be provided implicitly when a distribution is actually evaluated. Constant numbers are no longer allowed to define the distribution.
12. **Vector/Matrix** – Vectors and Matrices can hold arrays of numbers or parameters representing numbers. Vectors will have one dimension and matrices will be two dimensional. Their representation is very similar.
13. **State Indicators** – These parameters will represent states in the system. For each state created in the system, there will be 5 state indicator parameters with a similar name and a different postfix. The name of the parameters will be the same as the state where non alphanumeric characters will be replaced with underscore characters and a postfix of “Diagnosed” or “Treated” or “Complied” or “Entered” will be assigned. For example for the state “Survive MI” there will be 5 state indicator parameters called Survive_MI, Survive_MI_Diagnosed, Survive_MI_Treated, Survive_MI_Complied and Survive_MI_Entered . These parameters can be used in an expression during simulation and will represent the actual, diagnosed, treated, and entered states respectively. The value in the parameter associated with the state will be set to one if an individual is in this state in a simulation. It will be zero otherwise. For example, if for the state “Survive MI” the value of the parameters can be Survive_MI=1, Survive_MI_Entered=1, Survive_MI_Diagnosed=0, and Survive_MI_Treated=0 Survive_MI_Complied=0 meaning that the individual is in actual state “Survive MI” and has just entered it, while this is not the diagnosed or the treated state. For pooled states that represent sub-processes, this will mean that the state indicator parameter values will be set to one if the state associated with the sub-process and the individual will be set to one. For example, if “CVD” is a sub-process containing “Survive MI” . Then the values of the state indicators can be Survive_MI=1, Survive_MI_Entered=1, meaning as before that the individual actually entered the state of MI_Survive and therefore CVD=1. If CVD=0 this means that it has not been entered and therefore Survive_MI=0.
14. **Constant** – Similar to a function. Only constant. It is a general way the user can store constants and give them a name.
15. **System Options** – Parameters that their name is set by the system and can be modified by the user to change functionality of the system. These variables can be for example used as convergence criteria for optimization such as a tolerance.
16. **System Arrays** – Similar to system options, only containing a vector/matrix. May be used for example to define an initial set of default initial guesses for all parameters for an estimation project.

17. **System Reserved** – The parameter table in the Database may be used by the system to store temporary parameters to help in calculations and reserved names to avoid using them by other parameter names. For example 'Time' or 'IndividualID' can be reserved by the system. Also, internal functions names would be in the system reserved list so that a user will not use these by mistake. System reserved parameter that have formulas, will be allowed to be used in expressions as they will have a meaning defined by the formula, while system reserved words with an empty formula are banned for use. Banned system reserved words will help protect the system from using reserved words and functions of the language implementing the system.

Parameter names will be case sensitive. Note that the parameter "Age" and the parameter "AGE" are not the same. To help the user avoid case sensitivity mistakes whenever possible entry of parameter names will be performed in a combo box to help the user select the desired parameter.

Names of parameters should start with a letter and may not include spaces or special characters. However underscore is allowed. Formally, using a regular expression a parameter can be defined as:
Parameter = {a-z,A-Z}{a-z,A-Z,0-9,_ }*

Parameters will be generally created by the user. However, some parameters will be created by the system automatically. Such parameters include State Indicators, System Options, System Arrays, and System Reserved. In the case of state indicators, the parameters will have the name of a state or the same name of a state with one of the following suffixes: "_Diagnosed" or "_Treated" or "_Entered" or "_Complied".

The parameters will be stored as strings for input and output purposes and the software will internally convert these to and from the internal representation. Nevertheless, the string/value may be limited by the user and the system by using validation rules.

A validation rule will be applied to some parameters and functions. Validation rules can be of the following kinds:

- Number
    a. Can store a double precision number.
    b. Use of variable names within the formula string is not allowed, however an arithmetic operation between numbers can be used to provide a value of the parameter at input.
    c. In addition to the validation rule, validation rule parameters can be optionally be defined by the user. These define a range by providing two additional parameters to define a min and max bound to the values allowed. For example, for probabilities, these bounds can be defined as [0.0 , 1.0]. Note that the bounds are inclusive in this case. Infinity (Inf) and minus infinity (-Inf) can be defined as bounds. The validation rule parameters are represented as two numbers separated by a comma in between brackets stored as a string.

- Integer
    a. Can store an integer number.
    b. Use of variable names within the formula string is not allowed, however an arithmetic operation between numbers that results in an integer value of the parameter at input is allowed.
    c. In addition to the validation rule, validation rule parameters can be optionally be defined by the user. These define a range by providing two additional parameters to define a min and max bound to the values allowed. For example, for Booleans, these bounds can be defined as [0, 1]. Note that the bounds are inclusive. Note that the bounds are inclusive in this case. Infinity (Inf) and – infinity can be defined as bounds as well as real numbers. The validation rule parameters are represented as two numbers separated by a comma in between brackets stored as a string.
- Table
    a. Stores values, dimension names, and ranges for a multidimensional table.
    b. The table is described in a single string of the form "Table("+Parameters+")" where Parameters is a string of comma separated data that will hold the following set of values as a vector for numbers. First, the number of dimensions d will be defined, then the index limits for each dimension are defined $n_1, n_2, ..., n_d$. Finally, the values of the table are provided in a vector $\mathbf{v}$ of numbers . The members are indexed as follows: $i = i_1 n_2, ..., n_d + i_2 n_3, ..., n_d + ... i_d = \sum_{j=1}^{d} i_j \prod_{k=j+1}^{d} n_k$. Therefore the following vector will be stored to completely define the table and its values: $\mathbf{w} = d, n_1, n_2, ..., n_d, v_{i_1}, v_{i_2} ..., v_{i_{n_1 * n_2 * ... * n_d}}$. To allow easy access to the table, additional information is added at the end of this vector to define the table columns and their value ranges. First, the dimension name will be defined by giving the names m_k of a parameter such as Age or Gender that corresponds to dimension k where $1 \le k \le d$. Following the name, a list of numbers will define the ranges of values for the rows/columns corresponding with the parameter. Row/column values will be defined by range bounding values $p_{k,1}, p_{k,2}, ..., p_{k,n_k+1}$ that define the boundary between two consecutive row/columns. A range bounding value $p_{k,r}$ will separate between rows/columns in dimension k such that any entry in row/column r corresponds to bounding values where $p_{k,r} < Val(m_k) \le p_{k,r+1}$. For example if the table has columns for Age≤20, 20<Age≤50, Age>50 then this means there are 4 range bounding values: -∞,20,50,∞ for defining three columns. This type of representation also allows representing tables that are accessed by index like 1, 2, 3,.... In such a case the first bounding value will be set to NaN (not a number) to indicate this is an indexed table. Other bounding values will represent the values associated with the columns. This corresponds with the previous definition where the right bound is inclusive, i.e. in this case

$Val(m_k) = p_{k,r+1}$. To ease in reading this information, dimension names will precede the bounding values and each dimension will be supplied as a block following the table values:

$$\mathbf{w}^* = d, n_1, ..., n_d, v_{i_1}, ..., v_{i_{n_1 * n_2 * ... * n_d}}, \underbrace{m_1, p_{1,1}, ..., p_{1,n_1-1}}_{n_1}, \underbrace{m_2, p_{2,1}, ..., p_{2,n_2-1}}_{n_2}, ..., \underbrace{m_d, p_{d,1}, ..., p_{1,n_d-1}}_{n_d}$$

Table cells can hold expressions containing numbers and expressions that can contain coefficients and numbers. The validity check of the parameter string is as follows:

i. Number of dimensions is a positive integer
ii. All dimension indices / ranges are sorted in increasing order
iii. All table values are either numbers or coefficient parameter names
iv. Count of values correspond to multiples of all dimension sizes
v. The Dimension names are valid parameters that are one of the following types: Covariate, Intervention, Coefficient, or State Indicator, Function, System Reserved.
vi. The dimension names and ranges correspond to the defined number of index limits for each dimension.

- Matrix
  a. Stores values, of a one/two dimensional array that can represent a vector/matrix
  b. The matrix vector is represented by a string of the form similar to "[1,2,3]" for a vector or "[[1,2,3] , [4,5,6]]" for a matrix. In other words. The vector is built from a set of comma separated values within brackets where each value can be either a number, a parameter, or vector within brackets. If a parameter name is used, is should exist and must represent a number or an integer. Nested vectors should be of the same length.
- Expression:
  a. Can represent a general expression that uses other parameters
  b. Cannot return a matrix. See below for additional details on expressions.
- Distribution
  a. The distributions functions used are valid. Meaning:
     i. The expression has a valid syntax expression
     ii. Starts with a distribution function name
     iii. No other parameters used in it
     iv. If not empty, it is not a combination of functions, meaning the statistical function is the only function used in it
     v. If not empty, an additional argument can be added to the input variable list and the expression can still be calculated. This argument represents queried parameter and the distribution functions are designed to accept it as an optional parameter. In other words, this means the distribution function should be specified in random generator form rather than CDF form.

- When no validation rule is defined, it is assumed according to the table below. Parameter types will be associated by default with a specific validation rule and will be allowed to have a validation rule from a list of existing rules.

Type	Allowed Validation Rules **Default is highlighted**	Example	Comments
Covariate	**Number**, Number, Integer, Table, Expression	Age = 35	This is a very general data type. It can take many forms, including numbers expressions and tables. Used as Affected Parameters in simulation phase 1
Intervention	**Integer [0,1]** Number, Integer	FootExam =1	These are Booleans. Used as Affected Parameters in simulation phase 3
Cost	**Number [0,Inf]** Number [0,Inf]	Cost = 1549.9	These are positive numbers. A CostWizard can help calculate these. Used as Affected Parameters in simulation phase 4
Quality of Life	**Number [0,1]** Number	QoL=0.23	These are positive numbers. A CostWizard can help calculate these from a table. Used as Affected Parameters in simulation phase 4. Note that the default bounds are [0,1], yet a user may override this.
Prevalence	**Number [0,1]** Number [0,1]	PrevState1=0.2	The prevalence of a state in another pooled state. Note that an empty value is considered by the system as zero if used in a pooled state.
Probability	**Number [0,1]**	ProbFlipCoin = 0.5	A constant that can be used in the simulation form that represents a probability
Transition	**Number [0,1]**, Expression [0,1], Table [0,1]	Trans_State1_State2 = 0.23	Representing a transition probability between two states. Some validations are possible only in runtime
Coefficient	**Number** Number	P02=0.005	Generally, the value will be left empty. For unknown coefficients to be calculated during estimation. Actual

<table>
<tr><td></td><td></td><td></td><td>values can be assigned when Used as Affected Parameters in simulation phase 0</td></tr>
<tr><td>Function</td><td>Expression
Expression</td><td>Age+1</td><td>Holds general expressions that can be reused in other parameters</td></tr>
<tr><td>Table</td><td>Table
Table</td><td>Table(2,2,3,1,2,3,4,5,6, Gender, NaN,0,1, Age,0,30,60,120)</td><td>This represents the following table:
<table>
<tr><td></td><td>0<Age≤30</td><td>30<Age≤60</td><td>60<Age≤120</td></tr>
<tr><td>Gender = 0</td><td>1</td><td>2</td><td>3</td></tr>
<tr><td>Gender = 1</td><td>4</td><td>5</td><td>6</td></tr>
</table></td></tr>
<tr><td>Distribution</td><td>Distribution
Distribution</td><td>Gaussian (100,5)</td><td>Must be an expression representing distribution types. It may be used in population sets in estimation. The number/expression should use the distribution function without the last parameter, i.e. the function should be written in random generator form rather than as CDF form, although distributions use an implicit additional parameter to calculate a CDF.</td></tr>
<tr><td>Vector/Matrix</td><td>Matrix
Matrix</td><td>[[1,a], [a,2]]</td><td>This example represents the following:
<table>
<tr><td>1</td><td>a</td></tr>
<tr><td>a</td><td>2</td></tr>
</table>
Note that this kind of parameter cannot be used in many places. It is possible, however, to use it in expressions under some restrictions.</td></tr>
<tr><td>State Indicator</td><td>Integer [0,1]
Integer [0,1]

Also an associated state exists. There are some name</td><td>Angina=1</td><td>These are system generated variables that can be used in expressions and in simulation rules.
Note that state indicators may have the suffixes: _Entered,</td></tr>
</table>

	restrictions		_Diagnosed, _Treated , _Complied and the beginning corresponds to a state name.
Constant	**Number** Number Integer	1	General user defined constants
System Option	**Number** Number Integer	Tolerance = 0.0005	These numbers will be used to control system behavior such as defining some defaults and tolerances
System Array	**Matrix** Matrix	[0.1, 0.2]	Vectors to be used to control system behavior such as defining some defaults
System Reserved			These are system reserved variables. Some cannot be used by the user, this include python reserved words and some other banned names. Others can be used by the user and represent system functions such as CostWizard or some system defined variables such as Inf or IndividualID. These cannot be changed by the user

2.4. Dealing with Expressions

Expressions can contain a combination of numbers and other parameters such as covariates, tables, functions etc. These are primarily used in simulation and are defined in most cases by the user. Expressions are treated by the system in three phases:

1. Validation time – when the expression is first entered by the user. The following checks will be made:
    a. Verification that all tokens used in the expression exist and are either: numbers, operators, functions, or existing parameters. This is performed using the function $\mathbf{p} = \text{ExtractExprParam}(e)$ defined below
    b. Validation that there is no loop definition of any single parameter where a parameter recursively addresses itself. This is performed by recursively expanding parameters and searching of the parameters using the function $(r, \mathbf{p}) = \text{ConstructExprRecursively}(e, \mathbf{b})$ defined below
    c. Limited validation that the expression can be evaluated without expanding it with other parameters. This involves some lexical analysis and assigning mockup numbers to parameters and trying to evaluate the result and warn the user of possible hazards. This process is dependant on the implementation language. In the context of Python the following checks should be made after the expression tree is parsed:
        i. Verify that Python reserved words do not appear in the expression. This can be performed by adding the reserved words to the banned parameter list.
        ii. Confirm that the following symbols do not appear in the expression: ‘”~!@#$%^&:;?<>={} and //
        iii. The expressions will be checked for balanced parenthesis and other issues imposed by python syntax by using the python parser to parse the expression.
        iv. Corroborate that the expression contains no tuples that do not follow a name that corresponds to a parameter representing a function, i.e. the expression “(1,2,3)” is invalid, while “Func(1,2,3)” is valid if Func is defined as a non banned system reserved parameter. The list of valid functions is defined in the list below
        v. No operator, i.e. *,/,+,-,** is allowed before or after a list or an expression that results in a list (including lists in parenthesis). Note that lists can be input for functions and the output of a function is assumed to be not a list. For example “[1,2,3] + [1,2,3]” is not allowed.
        vi. The expression can be calculated without an error when:
            1. Parameters are replaced with numbers
            2. The division by zero error is overridden. One non-ideal way to do this is by replacing the division operator / with multiplication * to avoid division by zero. Another way is

catching this specific error. Note that a division by zero can still occur in runtime even if undetected by this check.

3. Functions are replaced with mockup functions that do not generate errors for faulty input. For example calling a table with an out of range index still returns a value, although meaningless. Note that this can be implemented by having a function with the same name for a validation time and for runtime. The validation time version of the function will perform only a simple number of variable checks and the runtime version will perform actual calculations.

vii. The expression is evaluated to a numeric value rather than a list or a tuple. Note that a list is allowed if the programmer specifically specifies that the expression results in a matrix, in which case only a matrix form will be allowed.

d. Limited validation that the expanded expression that recursively substituted other parameters can be evaluated properly. This again involves the same steps as checking the non-expanded expression.

2. Compilation time – when the system prepares the simulation program:
   a. All expressions to be used in all simulation instructions are checked again for validity as if in validation time. This is required as an assertion since the user may delete or change a parameter and by this invalidate another parameter that was already validated.
   b. Each expression is expanded by recursively replacing all known parameters with their formulas. Again, this is done using the following formula $(r, \mathbf{p}) = \text{ConstructExprRecursively}(e, \mathbf{b})$
   c. A simulation program is created by using the expanded expressions and supplementing these to a header that contains definitions of functions, and by creating a simulation control structure that handles loops and parameter initialization. Note that each parameter is expanded recursively if it contains other parameters and the nested parameters are checked to be valid within their validation rules as well as the topmost expression.

3. Run time – when the system runs the simulation program:
   a. The simulation program is executed.
   b. Results are imported back to the system. If a run time error occurred it is reported back to the system.

2.4.1. Supported Functions and Operators

Below is a list of built-in functions the system should support:

- Basic operations allowed for integers and floats.
  - A list of arithmetic functions:
    - + : Addition operator
    - - : negative/subtraction operator
    - * : multiplication operator
    - / : division operator (note that integers will be treated as floats)
    - ** : power operator
  - A list of comparison operators:
    - Eq(x1,x2): will return 1 if x1=x2 and 0 otherwise
    - Ne(x1,x2): will return 1 if x1<>x2 and 0 otherwise
    - Gr: will return 1 if x1>x2 and 0 otherwise
    - Ge: will return 1 if x1<=x2 and 0 otherwise
    - Ls: will return 1 if x1<x2 and 0 otherwise
    - Le: will return 1 if x1<=x2 and 0 otherwise
  - A list of Boolean operators:
    - Or (x1,x2,x3…): will perform a Boolean OR operation on two or more inputs
    - And (x1,x2,x3…): will perform a Boolean AND operation on two or more inputs
    - Not(x): will perform a Boolean Not operation on a single input
    - IsTrue(x): will return 1 for a numeric x that is not 0. Will return 0 otherwise.
  - A list of special math related functions:
    - IsInvalidNumber(x): will return 1 for x=NaN or for a non numeric type such as a vector , 0 otherwise
    - IsInfiniteNumber(x): will return 1 for x=-Inf or x=Inf, 0 otherwise
    - IsFiniteNumber(x): for will return 0 if x is not a valid number or an Infinite number, 1 otherwise

  Important note : Boolean operators treat NaN as false as well as any other non number type such as a vector/matrix.
- Mathematical functions
  - Exp(x)
  - Log(x,n)
  - Ln(x)
  - Log10(x)
  - Pow(x,n)
  - Sqrt(x)
  - Pi()

- Other functions:
  - Mod (x,n)
  - Abs(x)
  - Floor(x)
  - Ceil(x)
  - Max(a1,a2,a3…)
  - Min(b1,b2,b3…)
- Statistical Distributions – Random number generators (note the difference in argument number from the CDF form shown below):
  - Bernoulli (p)
  - Binomial (n,p)
  - Geometric (p)
  - Uniform (a,b)
  - Gaussian (mean,std)
- Statistical Distribution – CDF evaluation at point x (note the difference in argument number from the CDF form shown below):
  - Bernoulli (p,x)
  - Binomial (n,p,x)
  - Geometric (p,x)
  - Uniform (a,b,x)
  - Gaussian (mean,std,x)
- Control and DataAccess
  - Iif(Statement,TrueResult,FalseResult): Returns TrueResult if Statement <>0, FalseResult if Statement=0 and NaN if Statement is NaN
  - Table (Table Arguments) : Table arguments are provided as described previously. In simulation its parameters will be substituted.
- Application specific:
  - CostWizard (FunctionType, InitialValue, CoefficientVector, ValuesVector) : The function calculates the cost according the following formulas:
    - If FunctionType =0, meaning cost according to the Zhou paper $Output = InitialValue * 10^{CoefficientVector * ValuesVector}$
    - If FunctionType =1, meaning Quality of life according to the Zhou paper $Output = InitialValue + CoefficientVector * ValuesVector$

    Note that CoefficientVector and ValuesVector are vectors of the same size. And the system returns an error if there is incompatibility between parameters and coefficients or FunctionType is not 0 or 1.

Note that functions will return NaN in case of an argument value error, for example log of a negative number. However, in the case of Table and CostWizard, hard errors will be generated for some cases of invalid arguments. In contrast, a wrong number of input arguments will always cause a hard error. The user should be careful of using arguments in such functions so as not to cause a hard error during expression validation using substitution. Hard errors are generated on purpose since Table and CostWizard are not designed for variable input arguments in all fields. Users should be careful using variable arguments/expressions where these can generate errors using substitution validation.

2.4.2. Procedures for Dealing with Expressions

Extract Expression Parameters:

Name: $\mathbf{p}$ = ExtractExprParam(e)

Input:

- Expression string to be parsed: e

Output:

- A list of parameter $\mathbf{p} = \{p_i\}$ contained within the expression.

Algorithm:

1. Extract all tokens $\mathbf{x} = \{x_i\}$ in the expression using a tokenizing function available by the implementation language: $\mathbf{x} = \text{Tokentize}\{e\}$
2. For each $x_i \in \mathbf{x}$ perform the following:
   2.1. If x_i is a valid string
      2.1.1. If x_i is not a registered parameter then:
         2.1.1.1. Raise an error to the user indicating invalid input.
      2.1.2. Else, meaning that x_i is a registered parameter (note that functions and reserved words are considered system reserved parameters for this check) then:
         2.1.2.1. If the parameter is of type ‘system reserved’ and has an empty formula or the parameter is of type : 'Likelihood', raise an error telling the user that the parameter is not supported in this context.
         2.1.2.2. Otherwise, it is a valid parameter and therefore:
            2.1.2.2.1. $\mathbf{p} = \mathbf{p} \cup x_i$

Construct Expression Recursively

Name: $(r, \mathbf{p}) = \text{ConstructExprRecursively}(e, \mathbf{b})$

Input:

- Expression string to be parsed: e
- Banned parameter list: $\mathbf{b} = \{b_i\}$, if not defined, the default is an empty list.

Output:

- The expression with the replaced values: r
- A list of parameters $\mathbf{p} = \{p_i\}$ contained within the expression during the recursion. Duplicates are possible and the order is the order the parameters were traversed.

Algorithm:

1. Reset the output list of parameters $\mathbf{p} = \phi$
2. Reset the output list to the input expression $r = e$
3. Extract parameter names from the expression $\mathbf{a} = \text{ExtractExprParam}(e)$. Note that there are duplications in the name of the parameter and these are provided in the order of appearance. This is important to avoid problems later on in the algorithm.
4. For each $a_i \in \mathbf{a}$ traversed in the order of appearance, i.e. $i = 1..|\mathbf{a}|$ in the expression perform the following:
   4.1. If $a_i \in \mathbf{b}$ then:
      4.1.1. Raise an error indicating a recursive definition of parameter a_i. Note that raising an error stops the recursion. This can also be implemented by defining an error flag to the output of the function and exiting.
   4.2. Replace all the first occurrence of a_i in r with the string '$*i*$'. This provides unique identifier for each parameter so as not to confuse names later on when substitutions are made
5. For each $a_i \in \mathbf{a}$ traversed in the order of appearance, i.e. $i = 1..|\mathbf{a}|$ in the expression perform the following:
   5.1. If $a_i.Formula \neq \phi \wedge a_i.ParamType \neq$'System Reserved', meaning that the parameter has a formula and it is not a system reserved parameter:
      5.1.1. Recursively call the function to extract the parameter list $(x_i, \mathbf{q}_i) = \text{ConstructExprRecursively}(a_i.Formula, \{a_i, \mathbf{b}\})$
   5.2. Else, meaning that this parameter should not be replaces, and therefore:
      5.2.1. The parameter should appear in the text, i.e. $x_i = a_i$
      5.2.2. $\mathbf{q}_i = a_i$
   5.3. Replace the first occurrence of '$*i*$' in r with x_i
   5.4. Add the returned list of parameters to the existing list of parameters $\mathbf{p} = \mathbf{p} + \mathbf{q}_i$. Note that duplications are possible

3. Simulation

This section will briefly describe the simulation process from the developers point of view, emphasizing the algorithms used.

3.1. Initialization:

The system will gather all parameters needed for the simulation using these steps:

1. Verify that the model is valid by the following checks:
   1.1. Verify the definition of sub-processes
   1.2. In each sub-process, verify that the originating splitter has the same father as the current sub-process, meaning that there are no hierarchical jumps
   1.3. Verify there are no unconnected states in a sub-process that are not a starting point for the model.
   1.4. Verify that there is an obvious starting point for model/ sub-process. Note that this prohibits a pure loop – a started event state should be added to allow a pure loop.
   1.5. Verify that there is not more than one start point for the sub-process.
   1.6. Verify that the splitter state that generated all the sub-processes to be joined is the one the joiner state intends to join
   1.7. Verify that all event states and splitter states have at least one transition to exit them.
   1.8. Verify that all joiner states have exactly one transition to exit them.
   1.9. Verify the terminal end state is in the main sub-process.
   1.10. Raise a warning if a transition out from a splitter or joiner state has a probability defined that it is not 1.
2. Collect the parameter ‘Time’ that stands for the simulation time (in time units – usually years).
3. Collect the parameter ‘IndividualID’ that corresponds to the individual ID. Note that data uploaded in the population data should have a name different from this reserved name.
4. Collect the parameter ‘Repetition’ that will be the repetition counter.
5. Collect all parameters defined in the population set used in the project after it has been prepared for simulation in the following manner:
   5.1. Add all state indicators associated with the model in the project that do not already exist in the population set:
      5.1.1. Add all regular states that exist in the model and are not already defined and reset their values in all individuals to 0.
      5.1.2. Add all entered indicators to all the states that exist in the model and are not already defined and reset their values in all individuals to 0.
      5.1.3. Add all treated indicators to all the states that exist in the model and are not already defined and reset their values in all individuals to 0.

5.1.4. Add all diagnosed states that exist in the model and are not already defined and reset their values in all individuals to the value of the actual state indicator.
5.1.5. Add all complied indicators to all the states that exist in the model and are not already defined and reset their values in all individuals to 0.
5.2. For each state indicator set to 1, verify that all its ancestor sub-process indicators are set to 1. Note that this should be done for each individual for each category of indicators. i.e. actual, entered, diagnosed, treated, and complied.
5.3. For each sub-process set to 1 verify that there is only one child group set to 1. A child group is either a single non-sub-process state or all the sub-processes starting from the same splitter state. Again this is done for each individual and separately for each category of indicators.
5.4. Make sure that for each individual in the population at least one actual state indicator is set.
5.5. Make sure there are no individuals in the data with any parameter that is set to a non numerical value. If such are found, these are substituted with the mean value for this population set and a message will be delivered to the user.
6. All parameters that are defined in the project parameters as affected parameters

After gathering these parameters they will be updated in the following manner:

3.2. Main Simulation Algorithm:

1. Individual loop: For each individual from the population
1.1. Repetition loop: Repeat for the number of repetitions stated in the project
1.1.1. Reset parameters: All parameters are set to Zero before the simulation starts.
1.1.2. Initialize parameters:
1.1.2.1. Reset the current parameters vector according to the values provided in the population set data for this individual.
1.1.2.2. State indicators for diagnosed states are then copied from the actual state indicators since diagnosed states are considered to be the actual states.
1.1.2.3. Append the current parameters vector to the results to indicate the initial condition.
1.1.2.4. Reset the flag to terminate the simulation unless the individual is already in a terminal state at input, at which case set it.
1.1.3. Time Iteration loop: while the ‘Time’ is less than the time iteration limit stated in the project and the flag to terminate the simulation for the individual is not set, perform the following steps.
1.1.3.1. Increase the ‘Time’ iteration parameter by 1
1.1.3.2. Copy the current parameters vector from the previous parameters vector.

1.1.3.3. Stage 1: Covariate Update: Use the parameter update rules described below to change the covariates in the parameter vector according to the ordered set of rules setup in the project.
1.1.3.4. Stage 2: Update Complications:
    1.1.3.4.1. Reset all the state indicators corresponding to entered states.
    1.1.3.4.2. Reset all state indicators corresponding to sub processes that are not set. This includes Diagnosed/Treated versions of the processes. Note that resetting should be performed in hierarchical traversal order using the preorder scheme. This means that the main sub-process is handled first and sub-processes and their children are processed later. This way it is assured that a reset topmost process will force a reset of all states descending from it.
    1.1.3.4.3. Consult the simulation model and change the parameter vector according to the algorithm described below.
1.1.3.5. Stage 3: Intervention Update: Unless termination flag is up, use the parameter update rules described below to change the intervention parameters and the update diagnosed and treated state indicators in the parameter vector according to the ordered set of rules setup in the project.
1.1.3.6. Stage 4: Cost and QoL Update: Use the parameter update rules described below to change the Cost and QoL parameters in the parameter vector according to the ordered set of rules setup in the project.
1.1.3.7. Append the current parameters vector to the results table.

3.3. Parameter Update Rules

Parameter update rules are defined in each project for covariates in stage 1, for treatment parameters in stage 3 and for cost and QoL parameters in state 4. These rules are defined separately in each stage by the user. However, the computational mechanism used to interpret these rules is similar for all these parameter groups. The rules represent operations that affect different parameters in certain conditions and change them using a given calculation. The order of traversing the rules is defined by the stages and defined by the user within each stage.

Each rule consists of the following components:

Affected parameter: The parameter that its value will change. In each stage, different types of parameters are allowed to change: covariates in stage 1, treatment, diagnosed, and complied parameters and state indicators in stage 3, Cost and QoL parameters in Stage 4.

If in state: This conditional clause in the rule can contain any state indicator, including entered, diagnosed, and treated state indicators. The rule will be executed if the individual is currently in this state in the simulation, otherwise the affected parameter value will not

change. Note that the state indicator can represent a sub-process in which case the affected parameter will change if the individual is in this sub-process. Therefore, if the user wishes an affected parameter to be calculated disregarding the state it is in, the main sub-process of the model used in the project should be used, or the if clause can be blank, indicating no condition is set. Note that the same parameter can be affected by two or more rules having different 'if in state' clauses. Specifically, if two rules mention the same parameter and have two different 'if in state' clauses once with a state and once with a sub-process containing that state, then the parameter may be calculated twice. The order of calculation in such a case is defined by the order of the rules in the project.

Occurrence probability: This is another conditional clause that is similar to flipping a coin with a certain probability to decide if the affected parameter will be changed. A random number between 0 and 1 will be generated and compared to the expression provided in the occurrence probability clause. If the generated number is lower then the affected parameter will be evaluated, provided that the 'if in state' clause is true. Setting the expression in the occurrence probability clause to 1, means that this clause will always be true while setting it to zero means that the affected parameter will not change.

Formula: This parameter provides an expression that will update the affected parameter if the two conditional clauses are true. The expression can contain numbers, mathematical operator, and other parameters. It can also contain additional conditional clauses within the expression and may involve other parameters from various types such as tables, vectors and matrices. Tables will return the value from the cell that corresponds to all dimension parameters. Parameters will be evaluated by recursive substitution and while values will be substituted, the value will be validated against each substituted parameter type and user defined bounds. Bound checking will be performed for each substitution of a parameter and is recursive in nature.

3.4. Update Complications:

The following operations are performed according to the simulation model selected in the project. Changes are recorded on the current parameters vector and in the results table. Note that this procedure may raise a flag that will stop simulation for an individual due to death.

During simulation the system will hold a State Processing Queue (SPQ) that will hold all the states that are yet to be processed by the system. States in this vector will be traversed in the order they are written in the vector. In the vector, states will be ordered as groups in the following order of precedence: Terminal end states, splitter states, joiner states, event states, regular states, and sub process states. Note that a terminal end state is defined as a non-sub-process state with no outbound transitions that belongs to the main process of the model.

1. Collect all the actual (non-sub-process) states the individual is in from the current parameter vector and put them in the SPQ.

2. Reorder the SPQ according to their priority group while adding small random permutation to reorder states within the priority group.
3. While the SPQ is not empty, remove the first state from the start of the list and perform the following for this state:
   3.1. If the processed state is a terminal end state in the main sub process:
      3.1.1. Raise the terminal state flag
      3.1.2. Exit updating complications by clearing the SPQ.
   3.2. If the processed state is a splitter state:
      3.2.1. For each new state emanating from the splitter state perform the following:
         3.2.1.1. Reset the splitter state indicator
         3.2.1.2. Set the new state indicator in the parameters vector
         3.2.1.3. Set the new state indicator for entered in the parameters vector
         3.2.1.4. Set the new sub process indicator associated with the new state in the parameters vector
         3.2.1.5. Set the new sub process indicator for entered associated with the new state in the parameters vector
         3.2.1.6. If the new state is not a regular state, add the new state to the SPQ sorted according to its priority group. This means, splits before joiner states, joiner states before events, events before regular states etc. Within each group the state will be sorted according to the random order indicated by the process as defined by the initiating splitter state.
   3.3. If the processed state is a regular state, an event state, or a joiner state:
      3.3.1. If the state is a joiner state:
         3.3.1.1. For each sub process that the joiner state joins perform the following:
            3.3.1.1.1. For each active state that is a descendant child of this sub process, including children, grand children etc., perform the following:
               3.3.1.1.1.1. If the state is in the SPQ, remove it from there.
            3.3.1.1.2. Reset the sub process and all its nested sub processes from the state indicator in the current parameters vector. Note that this will require finding all sub-processes recursively nested within the sub-process.
      3.3.2. Consult the transition graph of the model and determine the chances to stay in the state and the chances to transit to other states. This may require calculation of a formula that depends on current parameters. Note that while calculating the formula, recursive substitution of parameters may occur and each substitution may carry a parameter type check and a bound check. The final value is checked to be within the [0,1] domain.
      3.3.3. If the state is an event state or a joiner state:
         3.3.3.1. Make sure that the sum of all transition probabilities emanating from the state sum to 1
      3.3.4. Generate a random number and determine which will be the next state.
      3.3.5. If a transition to another state occurs:
         3.3.5.1. If the next state is a terminal end state in the main process, an event state, splitter state, or joiner state add this state at the beginning of the

SPQ to be processed first in this time iteration. This can be achieved by setting the priority to the lowest priory possible before adding to the SPQ.

3.3.5.2. Reset the old state indicator in the parameter vector.

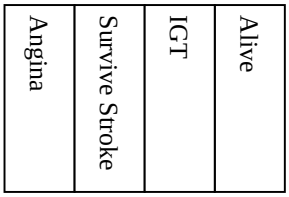
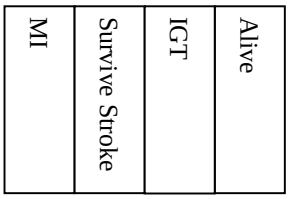
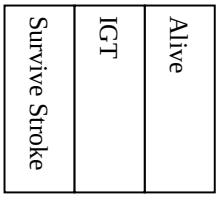
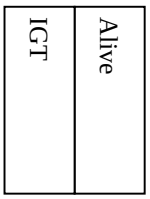
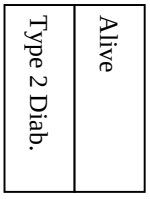
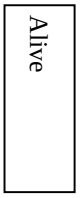
3.3.5.3. Set the new state indicator in the current parameters vector

3.3.5.4. Set the new state indicator for entered in the current parameters vector

3.4. If the processed state is a sub-process: Do nothing

3.5. Simulation Examples

The following figures show examples of the progress of the above simulation algorithms:

Current Parameters Vector	**SPQ**
Start: The SPQ was loaded in random order from the states in the parameters vector that show: States: Angina=1, Survive_Stroke=1, IGT=1, Alive=1, Sub-processes: Impared_Gloucose=1, Cardiovascular_Disease =1, Cerebrovascular_Disease=1, Competing_Death=1, Main_Subprocess=1	
Process Angina: Transition to MI. The following changes take place in the current parameters vector: States: Angina=0, MI=1, MI_Entered=1	
Process MI: Transition to MI history that is added to the SPQ. The following changes take place in the current parameters vector: States: MI=0, MI_History=1, MI_History_Entered=1	
Process Survive stroke: No transition. Therefore no changes take place in the current parameters vector.	
Process IGT: Transition to the splitter state - Type 2 Diabetes that is added to the SPQ. The following changes take place in the current parameters vector: States: IGT=0, Type_2_Diabetes=1, Type_2_Diabetes_Entered=1	
Process the state Type 2 Diabetes: The state splits to 3 sub processes and their start states. The following changes take place in the current parameters vector: States: Type_2_Diabetes=0, Treatment_Diet=1, No_Neuropathy=1, No_Nephropathy=1, No_Retinopathy=1, Treatment_Diet_Entered=1, No_Neuropathy_Entered =1, No_Nephropathy_Entered =1, No_Retinopathy_Entered =1 Sub-processes: Type_2_Diabetes_Process = 1, Neuropathy =1, Nephropathy=1, Retinopathy=1, Type_2_Diabetes_Process_Entered= 1, Neuropathy_Entered =1, Nephropathy_Entered =1, Retinopathy_Entered =1	
Process the state 'Alive': No transition and therefore no changes take place in the current parameters vector. Since the SPQ is now empty, processing complications will not continue in this time iteration.	Empty list

Example 1 – Processing Event and Splitter states

Current Parameters Vector	SPQ
Start: The SPQ was loaded in random order from the states in the parameters vector that show: States: Insulin=1, Amputation=1, ESRD=1, Alive=1 No_Retinopathy=1, No_CVD=1, Survive_Stroke=1 Sub-processes: Impared_Gloucose=1, Cardiovascular_Disease=1, Cerebrovascular_Disease=1, Competing_Death=1, Type_2_Diabetes_Process=1, Neuropathy=1, Nephropathy=1, Retinopathy=1, Type_2_Diabetes_Process_Entered=1, Neuropathy_Entered=1, Nephropathy_Entered=1, Retinopathy_Entered=1, Main_Subprocess=1	Insulin, ESRD, No Ret., Alive, No CVD, Amputation, Survive Stroke
Process Insulin: No transition. Therefore no changes take place in the parameters vector.	ESRD, No Ret., Alive, No CVD, Amputation, Survive Stroke
Process ESRD: Transition to ESRD Death that is added to the SPQ since it is an event state. The following changes take place in the parameters vector: States: ESRD=0, ESRD_Death=1, ESRD_Death_Entered=1	ESRD Death, No Ret., Alive, No CVD, Amputation, Survive Stroke
Process ESRD Death: Transition to Diabetes Death that is added to the SPQ since it is an event state and a joiner state. The following changes take place in the parameters vector: States:ESRD_Death=0, Diabetes_Death_Entered=1, Diabetes_Death=1	Diabetes Death, No Ret., Alive, No CVD, Amputation, Survive Stroke
Process Diabetes Death: Since this is a joiner state all the states in the sub processes it joins are removed from the SPQ In this case No.Rep. and Amputation. Also the following occurs: Sub-processes: Type_2_Diabetes_Process=0, Neuropathy=0, Nephropathy=0, Retinopathy=0. Since this is an event state as well that leads to death, Death is added to the SPQ and the following also occurs: Diabetes_Death=0, Death=1, Death_Entered=1	Death, Alive, No CVD, Survive Stroke
Process Death: Since this is a joiner state all the states in the sub processes it joins are removed from the SPQ. In this case Alive, NO CVD, and Survive Stroke. Also the following occurs: Sub-processes: Impared_Gloucose=0, Cardiovascular_Disease=0, Cerebrovascular_Disease=0, Competing_Death=0. Since this is a terminal end state in the main sub process, empty the SPQ and raise the terminal state flag. This will stop the simulation.	Empty list

Example 2 – Processing Joiner and Terminal States

Simulation Output Example:

		Covariates					Treatment/ Evaluation/ Intervention			States and Sub-processes parameters (Actual)										States and Sub-processes parameters (Entered)										States and Sub-processes parameters (Diagnosed)										States and Sub-processes parameters (Treated)									
Individual ID	Time Iteration	Age	Gender	Race	Blood Pressure	…	Ace Inhibitor	Foot Exam	…	Model	No Diabetes	IGT	Type 2 Diabetes	CVD	No CVD	Angina	MI	Survive MI	…	Model	No Diabetes	IGT	Type 2 Diabetes	CVD	No CVD	Angina	MI	Survive MI	…	Model	No Diabetes	IGT	Type 2 Diabetes	CVD	No CVD	Angina	MI	Survive MI	…	Model	No Diabetes	IGT	Type 2 Diabetes	CVD	No CVD	Angina	MI	Survive MI	…
1	0	34	0	2	130	…			…	1			1						…	1			1						…	1									…	1			1						…
1	1	35	0	2	132	…			…	1				1	1				…					1	1				…	1									…	1			1						…
1	2	36	0	2	135	…			…	1				1		1			…							1			…	1			1						…	1			1						…
1	3	37	0	2	147	…			…	1				1			1	1	…								1	1	…	1			1						…	1			1						…
1	4	38	0	2	142	…			…	1				1				1	…										…	1				1				1	…	1				1	1				…
1	5	39	0	2	153	…			…	1				1			1	1	…								1	1	…	1				1				1	…	1				1			1	…
1	6	40	0	2	132	…			…	1				1				1	…										…	1				1				1	…	1				1			1	…
2	0	55	1	1	123	…			…	1	1								…	1	1								…	1	1								…	1	1								…
2	1	56	1	1	142	…		1	…	1		1							…			1							…	1		1							…	1		1							…
2	2	57	1	1	124	…	1		…	1			1						…				1						…	1			1						…	1			1						…
2	3	58	1	1	132	…			…	1			1						…										…	1			1						…	1			1						…
2	4	59	1	1	132	…			…	1			1						…										…	1			1						…	1			1						…
2	5	60	1	1	135	…			…	1				1	1				…					1	1				…	1				1	1				…	1				1	1				…
2	6	61	1	1	138	…			…	1				1		1			…							1			…	1				1		1			…	1				1		1			…
2	7	62	1	1	129	…			…	1				1			1		…								1		…	1				1			1		…	1				1			1		…

Patient diagnosed with MI history after MI event

Patient was not treated the first step

Only entry to a state is marked here

The main process is always on

The user should be able to extract a single line from this table for a specific repetition to create a new population set.

4. Report Engine

The system will provide a Report engine with the following capabilities:

1. Each major entity in the system will be able to generate a report describing its data. This includes collections.
2. The report will be textual in nature.
3. Each report will be able to accommodate input parameters
4. Here is a list of parameters used by the system:

   Parameters to be used by any report:

   a. Detail Level: a number representing how much information will be displayed in the report. The default is 0.
   b. Show Dependency: if True the report will contain additional information regarding dependencies between entities, such as states and the associated state indicators etc. In addition, some expressions will be explained in a more readable fashion to the user.

   Parameters to be used by Simulation Results report:

   c. Summation Intervals: This will be a number, a pair in brackets or a sequence of numbers/pairs in brackets separated by commas. Pairs in brackets will represent a start/end of an interval with summary information whereas numbers will represent summary interval lengths in simulation steps that will be interpreted to summary information. If the number 0 appears as a number, this will mean that interval length will start applying at step 0, which includes initialization data. The system will always add the overall interval starting at the beginning, which is either 0 or 1 depending if 0 appears and ends with the maximum simulation step. The default for this will be [[[0,0],1,5,10,20] + the largest interval.
   d. Integer Number format: A string that will decide the number format of integer columns of the report. Python formatting convention is used. The default is %i.
   e. Float Number format: A string that will decide the number format of floating point columns of the report. Python formatting convention is used. With the addition of defining the %v option. When %(GuarenteedLeadSpaces).(DesiredPrecisionDigits)v is defined, then the precision of the number will be defined by the maximal value encountered for this column and the DesiredPrecisionDigits requested. This means that DesiredPrecisionDigits defines precision relative to the longest number to be written. If the length of the longest integer is longer than this number, then precision is zero digits after the period. Note that this essentially keeps only significant digits displayed. GuarenteedLeadSpaces defines a guaranteed number of spaces the number will take unless the Integer part becomes longer than this number. Note that if capital V is used, than the maximal number is selected from all categories of display, meaning that sums and column numbers are entwined when deciding on precision, otherwise each category is considered separate for the sake of deciding on

precision. The default float number format is %0.5v. If just %v is defined, it is the same as %0.5v.

f. Column Separator: This parameter can also be referred to as Column Spacing and it defines the characters that separate report columns in the text. This is especially helpful to define a column delimiter for importing the text to another application. The default is ‘ | ‘ , i.e. a vertical line with two space characters around it.
g. Selected Columns: This parameters is actually a Column Filter that defines which columns to display in the report, in what order and how to display each column. Note that repetitions of the same column in different positions or different characteristics is possible. The default is an empty list, meaning that all columns will be displayed according to the default options. This parameter consists of a list of structures with the following information:
    i. Column Name / Group Name: This is a string that is either the name of a parameter or a string representing a Group of parameters. Valid parameter groups are: '<Covariate>', '<Intervention>', '<Cost>', '<Quality of Life>', '<State Indicator>', '<System Reserved>', '<State Indicator,State>', '<State Indicator,State_Actual>', '<State Indicator,State_Entered>', '<State Indicator,State_Diagnosed>', '<State Indicator,State_Treated>', '<State Indicator,State_Complied>', '<State Indicator,Sub-Process>', '<State Indicator,Sub-Process_Actual>', '<State Indicator,Sub-Process_Entered>', '<State Indicator,Sub-Process_Diagnosed>', '<State Indicator,Sub-Process_Treated>', '<State Indicator,Sub-Process_Complied>', '<Header>'
    ii. Column Title: This parameter holds a string that if it is not empty it will replace the column head title in the text for raw data. An empty string indicates no change in parameter heading.
    iii. Summary Calculation Method: This option defines how the data will be displayed in the summary line. Options are:
        1. None – No information will be displayed for this parameter at the summary line. This is the default for system Reserved Parameters such as IndividualID, Time, and Repetition.
        2. Sum over all records – will sum values from all the records in the summary interval. This is the default option for Booleans that are not demographics, e.g. State indicators.
        3. Sum over demographics – will sum values from records of individuals entering the interval only. This is the default option for Booleans that are unaffected by the simulation i.e. non state indicators not in the affected list of the simulation project.
        4. Average over all records – will average values from all the records in the summary interval. It is equivalent to dividing the sum over all records by the total number of all records

in the interval. This is the default option for non-Booleans that are not demographics, i.e. parameters that may be affected during the simulation.

5. Average over demographics – will average values from the records entering the interval only. It is equivalent to dividing the sum over demographics by the total number of records entering the interval. This is the default option for non-Boolean parameters that are in the affected list of the simulation.
6. Total number of records – Return the total number of records in the interval. In a sense this ignores the parameter itself and used by system defaults.
7. Demographic count – Returns number of records entering the interval. In a sense this ignores the parameter itself.
8. Interval Start – return the Simulation start step number of the interval. In a sense this ignores the parameter itself.
9. Interval End – return the Simulation end step of the interval. In a sense this ignores the parameter itself.
10. Interval Length – return the number of steps represented in the interval. In a sense this ignores the parameter itself.
11. Sum Over Last Observations Carried Forward – This calculation option will sum the carried forward column values for all the records starting from the beginning of simulation and ending at the summary interval end. Carried forward means that missing records due to premature termination of simulation for an individual are replaced by the last value for column for this individual.
12. Average Over Last Observations Carried Forward – This calculation option will average the carried forward column values for all the records starting from the beginning of simulation and ending at the summary interval end. Carried forward means that missing records due to premature termination of simulation for an individual are replaced by the last value for column for this individual. This option may be useful to average cumulative parameters such as total cost parameters for an individual.
13. Automatically detect – This is the default option where the system will select the default according to the parameter in question as described above.

5. Special attention will be provided to the simulation results report that will contain summary information at the end of the report. First the column headings will be defined that will define the summary method. This will include a line for each interval with the header before any other columns with summary information for each column will follow according to the parameters presented before. Note that this Header will appear by default and can be removed by the user if columns are

selected without selecting the '<Header>' group. The header will specify the following automatically generated columns:

a. Interval Start Step
b. Interval End Step
c. Demographic count, i.e. the number of records entering the Interval
d. The Total number of records - for this interval

5. Graphic User Interface (GUI) Framework

This chapter will briefly overview the structure of the Graphic User Interface (GUI) of the system. The GUI concepts are briefly described hereby in a nutshell. Some issues have been simplified and the reader is directed to the code for up to date additional information.

5.1. Terminology

5.1.1. Terminology and Concept of GUI

- Form : Form is a combined concept of viewable entities and control algorithm. Physically, Form is a python script file that contains two or more class. MainFrame class, RowPanel class and several methods should be implemented in those class. Common control algorithm is implemented in CDMLib.py as a method of CDMFrame class.

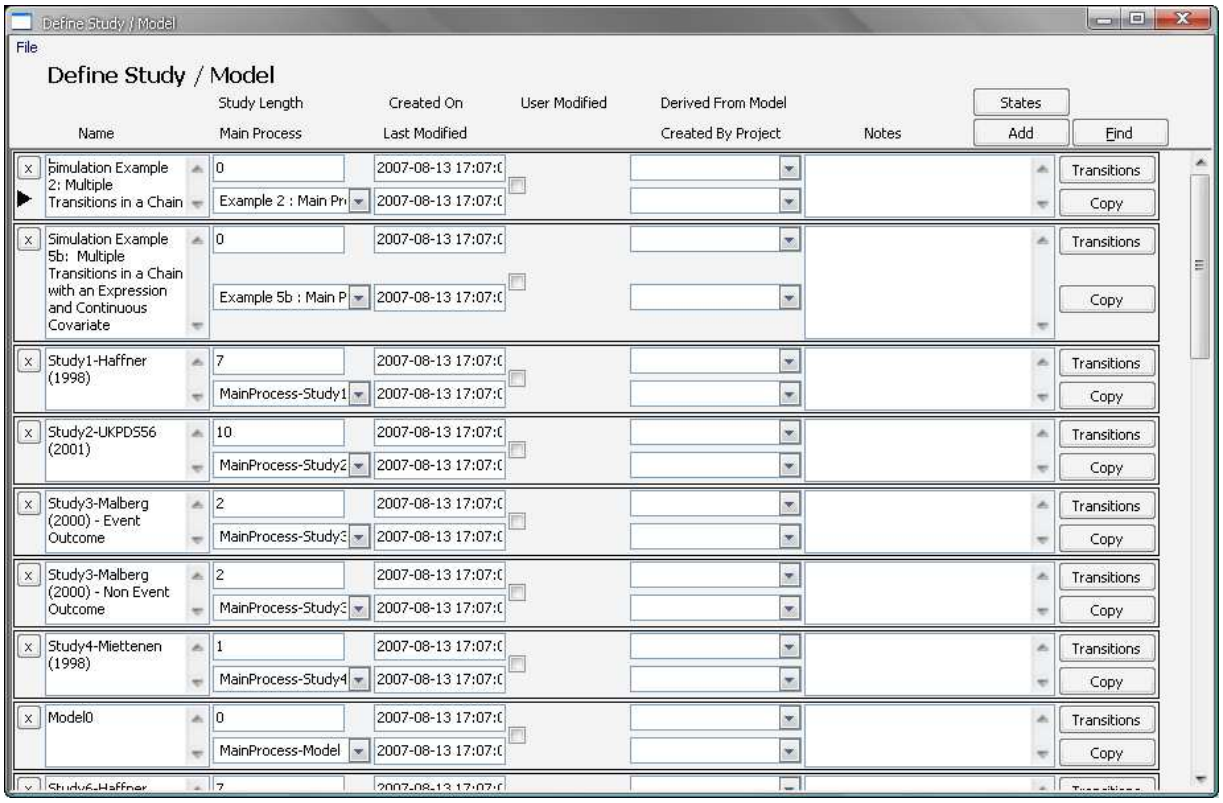


Figure. Sample screen shot of a Form

- MainFrame : External window except RowPanel will be called as MainFrame. Usually, it includes title of data fields, buttons and a scrolled window control that hold RowPanels. Some frame may have menus.

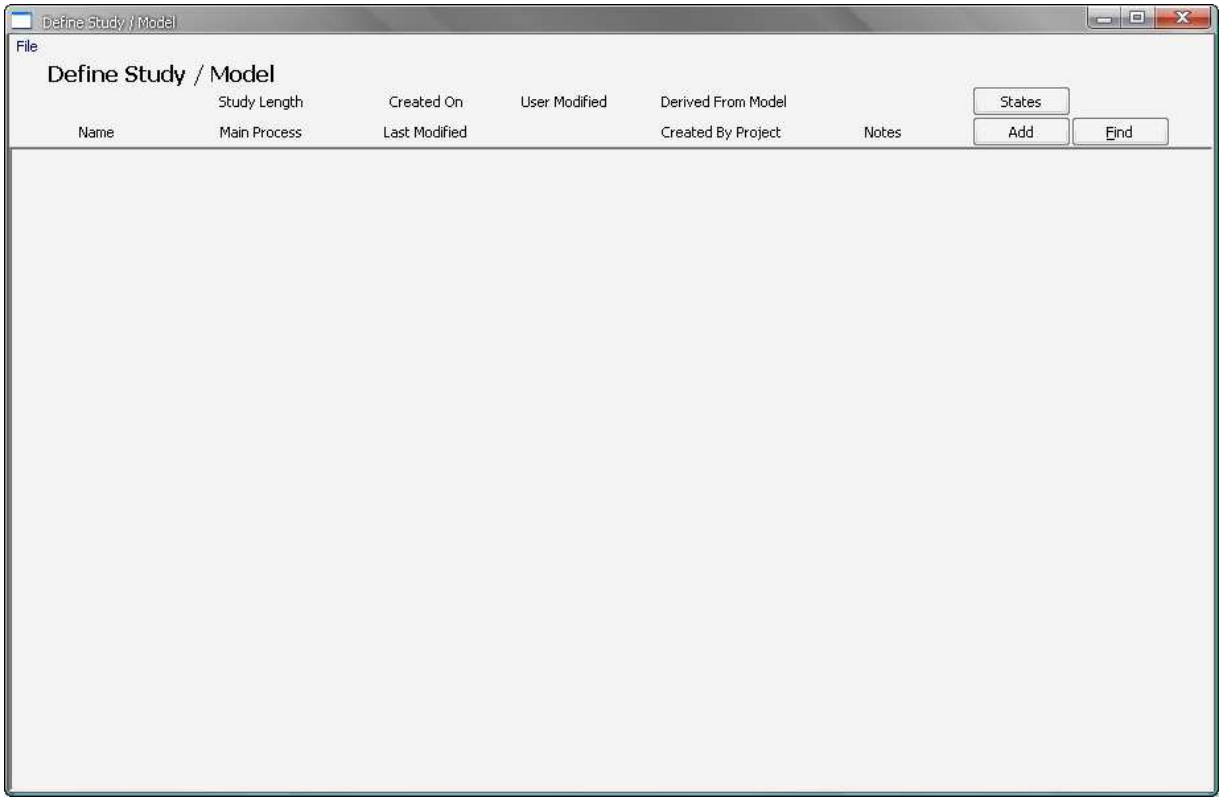


Figure. Sample screen shot of a MainFrame

- RowPanel : RowPanel is the place for data entry and modification. Each instance of RowPanel class has one-to-one relation with object in Database.

Figure 1. Sample screen shot of a RowPanel

Simply speaking:
Form = MainFrame + RowPanels + Functions and Event Handlers

5.1.2. What consists of a Form?

The following Figure describes elements that build a form.

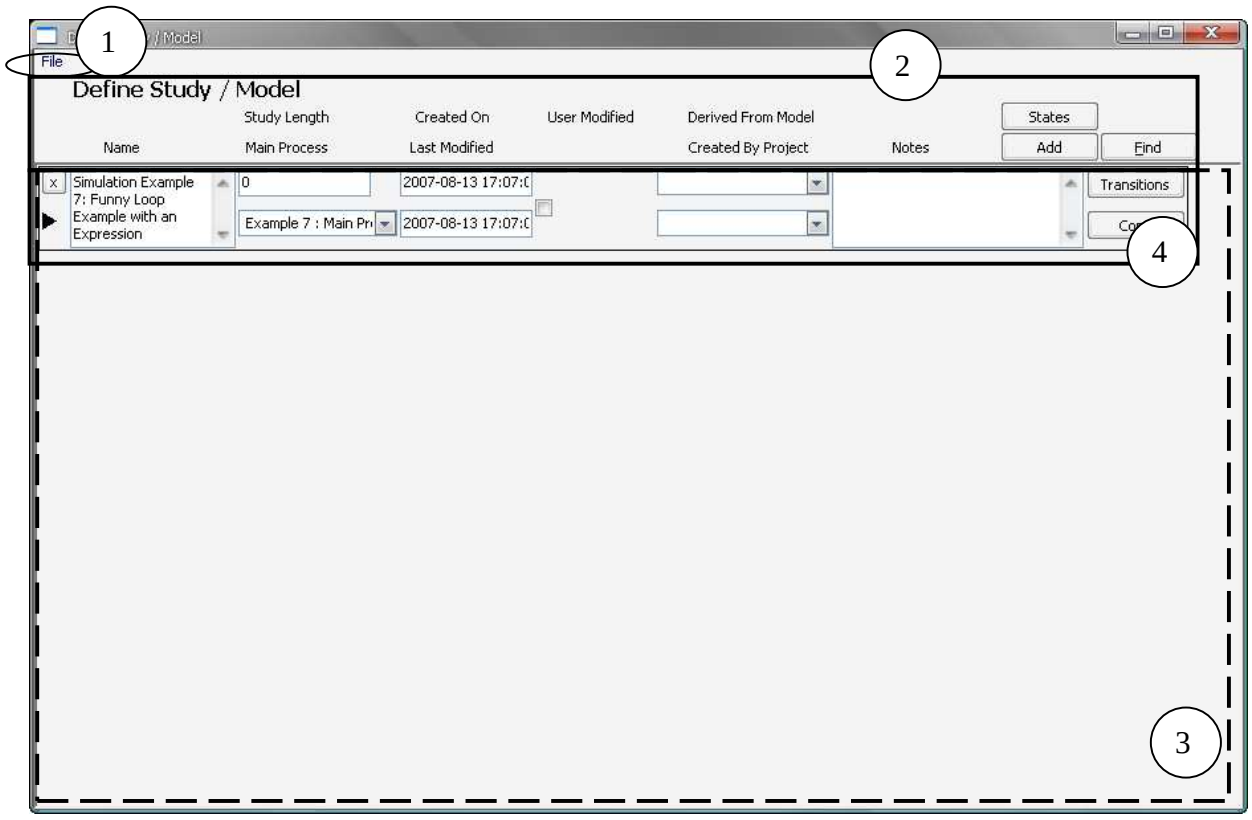


Figure. Sections in a Form

① Menu : Some forms don't have menu
② Title Section : Include title of each field in a RowPanel and buttons for action
③ Scrolled Window : Window that the RowPanels are placed.
④ RowPanel : A class to display/edit database object. Each RowPanel instance need to be matched to an object in a collection

5.1.3. File Structure

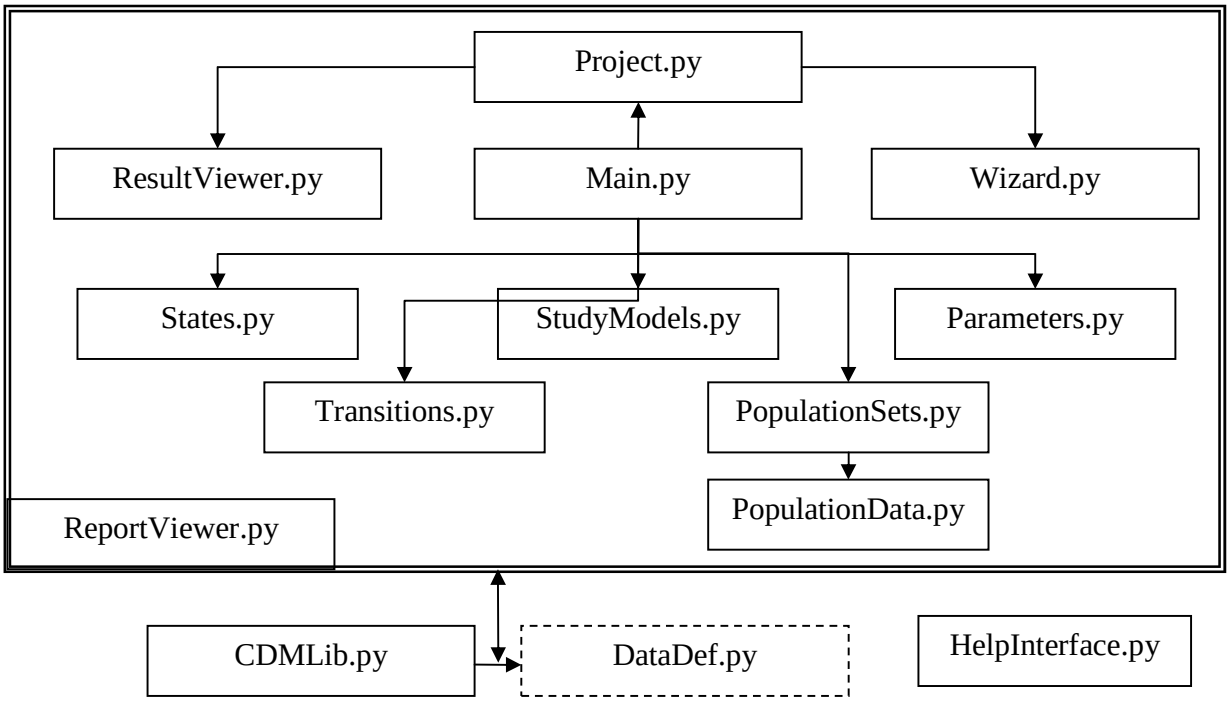


Table. Modules and its functions (Extension is omitted)

Name	Comment
Main	Main form
States	Data entry/modification for States
Transitions	Data entry/modification for Transitions
StudyModels	Data entry/modification for StudyModels
PopulationSets PopulationData	Data entry/modification for PopulationSets
Parameters	Data entry/modification for all Parameters (including code generated parameters) in database
Project	Data entry/modification for a Project and simulation running
Wizard	The cost/QoL Wizard used by the project
ReportViewer	Shows simulation results for the project
ReportViewer	Show a report and define report parameters
HelpInterface	Functions to invoke the help system
CDMLib	Includes all common classes, functions and constants used in the GUI

5.2. How to Create a Form

The major function of the GUI is to manage functions for data entry/maintenance process. Thus, the process to add new form for a collection will be presented in this chapter. However, there may be some exceptions according to the functionality of a form. Very common procedures will be introduced in this document using 'StudyModels' form as an example form.

The following table shows the common procedure to add new form. Same procedure can be applied for both MainFrame and RowPanel class.

Table. General procedure to create new form

Step	What to do?
1	Create new MainFrame class by subclassing from CDMFrame class
2	Add controls to the MainFrame class
3	Set Frame/Controls properties and positions
4	Bind event handlers
5	Implement mandatory/additional methods if necessary

Details in each step are shown in next section step by step.

5.2.1. Create New MainFrame Class

Current version of GUI uses subclassing to create new frame. Thus, all frames that need to manage objects (or collection) should be instanced from CDMFrame class and should be named as 'MainFrame'. The frame creation process consists of three methods (see code below).

```
class MainFrame(cdml.CDMFrame):
    def __init__(self, mode=None, data=None, type=None,
                 id_prj=0,*args,**kwds):
        : initialize CDMFrame class
        : create controls
        : call __set__properties and __do_layout methods

    def __set__properties(self):
        : set properties of MainFrame and controls
        : ex) size, font, event data, etc

    def __do_layout(self):
        : set the position of controls
```

Default methods to create a new form

__init__ (self, mode=None, data=None, type=None, id_prj=0, *args, **kwds):

When a class is called to create new instance, __init__ method is called at first. Thus, all class have to have __init__ method. __set_properties and __do_layout method can be included in the __init__ method, separate methods will be used in current version

__init__method includes most creation methods for conrols. Event handler assignment is also done in this method.

__set_properties (self):

As can be expected, properties of frame and controls are set in this method

__do_layout (self):

This method is used for the positioning of controls.

- Create a frame for new form
  - Call __init__ method of CDMFrame class
  - Syntax : cdml.CDMFrame.__init__(self, mode, data, type, *args, **kwds)
  - You may not need to change the syntax for all frame which is derived from CDMFrame class
  - States, StudyModels, PopulationSets, Transitions and Parameters forms are in this category

- PopulationData, Project, SimulationResults, and ReportViewer forms don't need to manage panels. Thus, those forms are instanced from wx.Frame class

* cdml is defined at the top of the module : import CDMLib as cdml

5.2.2. Add Controls to the MainFrame Class

After creating the frame(external window for a form), controls to enter/edit the DB fields should be created.

1. Add menu bar and menu items if necessary
   - See MainFrame class in module 'StudyModels'
   - Menus are used in a mostly generic method in the application with a few exceptions. Generally they are used to
     - Save/load files from the main menu
     - Call the report engine
     - Call the help interface
     - Add new record, copy record and undo changes from the popup up menu

2. Create a panel for the title section.
   - Syntax : cdml.CDMPanel(isRow, parent, id)
     - dx : `self.pn_title = cdml.CDMPanel(False, self, -1)`
   - The name of panel should be 'pn_title'
   - First argument(isRow) should be False always

3. Create title if necessary
   - Syntax : wx.StaticText(parent, id, title_label)
     - ex : `self.st_title = wx.StaticText(self.pn_title, -1, "Define Study/Model")`
   - First argument is the name of parent. So, it should be 'self.pn_title' for all controls in title section

3. Create buttons for field name in a database class
   - Syntax : cdml.BitmapButton(parent, id, bitmap, label)
     - ex : `self.button_3 = cdml.BitmapButton(self.pn_title, cdml.IDF_BUTTON1, None, "Name")`
   - All buttons for database field name should an instance of cdml.BitmapButton class and specific id should be assigned for each button

4. Create button to add RowPanel instances and button for find function
   - Syntax : `.Button(parent, id, label)`
   - ex :
     `self.btn_add = cdml.Button(self.pn_title, wx.ID_ADD)`

```
self.btn_find = cdml.Button(self.pn_title, wx.ID_FIND)
```

- The id should be wx.ID_ADD and wx.ID_FIND. These ids are used in DefaultEventHandler
- When these ids are used, you don't need to set label. The labels are automatically set by wxPython.

5. Create an instance of ScrolledWindow class that instances of RowPanel class will be placed.
   - Syntax : `wx.ScrolledWindow` (parent, id, style)
     - ex : `self.pn_view = wx.ScrolledWindow(self, -1, style=…)`
   - No restrictions exist about the style. However, wx.TAB_TRAVERSAL style needs to be set for 'Tab' key

5.2.3. Set Frame/Controls Properties and Position of controls

Set Frame/Controls Properties

1. Set title and size
   - Syntax : `self.SetTile` (TitleLabel)
   - Displays the title of frame on the title bar

2. Set the name of global variable which is related to this form
   - Syntax : `self.SetCollection(VariableName)`
   - VariableName should be a name defined in 'GlobalsInDB' list in the DataDef module

3. Set other important variables
   - Set the help context string `self.HelpContext` to allow calling the help interface in a generic manner. The call is channeled through the menu system through cdmlib.py using HelpInterface.py. For this reason, make sure the help context corresponds to the appropriate page to be loaded. If not defined the general help page will be loaded instead.
   - Set the current Project id to be used. Usually, this is called by using the line `self.idPrj = id_prj` .This is done to allow drill down operations from a specific project and may not be required in some cases.

4. Set the sort id for field titles
   - The sort id should be assigned to each field title(i.e. bitmap button) and it should be matched with the sort id assigned to controls in an instance of RowPanel class.
   - When user clicks the label in the title section, the Sort function uses this id to recognize the target field. (See Figure below)
   - evtID for sort function should be ID_EVT_SORT

5. Set the event data for additional buttons (See buttons at upper right corner in Figure below)
   - Sometimes additional buttons need to be added in the title section. To check the focus change to these buttons from RowPanel instances, the GUI supply special event handling mechanism.
   - Syntax : `SetEvent(( evtType, evtID, evtData))`
     evtType : Define event type. In current version, this argument isn't used.
     evtID : To assign specific event handler, this argument should be ID_EVT_OWN
     evtData : name of event handler for this button (or control)

Set Position of controls

- Positioning of controls is done in the __do_layout method using Sizers
- For simplicity and convenience, GridBagSizer is most frequently used in current version.

- In this manual, detail description about how to use the sizer will not be presented. See the wxPython manual and this link (http://neume.sourceforge.net/sizerdemo/) for details about the sizer.

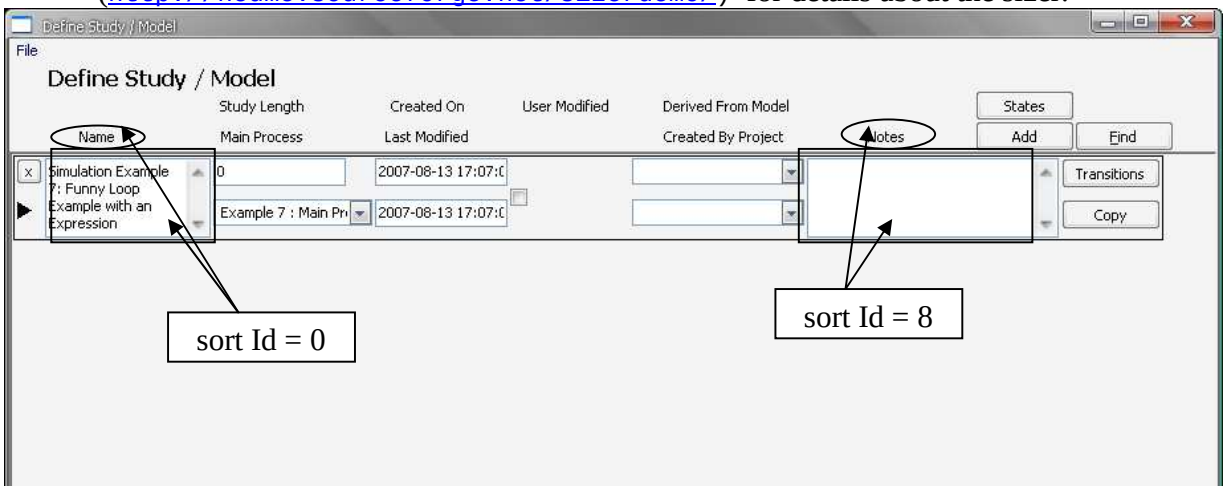


Figure. Set sortId

5.2.4. Bind Event Handlers : Default and Specific Handlers if Necessary

- All controls need to check focus change should be bound to DefaultEventHandler.
- The method for'Add' and 'Find' buttons are implemented in the CDMLib module. Thus, these buttons also need to be bound to DefaultEventHandler
- All other controls which need to call own event handler should use event properties. Nonetheless, the event handler should be implemented in the MainFrame class

5.2.5. Implement Additional Methods if Necessary

- Here, simple example is shown for the 'Define States' button in the title section. Following is the sample event handler for the button.

```
def OpenFormStates(self, event):
     """ Event handler to open child window """
     cdml.OpenForm('States', self, None, None)
```

Example of more methods

5.3. How to Create New RowPanel Class

Procedure to create RowPanel class is shown in the Table below.

Table. General procedure to create new form

Step	What need to do?
1	Create new RowPanel class by subclassing from CDMPanel class
2	Add controls to display/edit field value
3	Set Frame/Controls properties and positions
4	Bind event handlers
5	Implement mandatory/additional methods

First four steps are much similar to those of MainFrame class. Thus, detail descriptions will not be given in this section (However, some detail items will be pointed). Because most actions for the data entry/modification are performed in the RowPanel class, much effort will be given to explain Step 5.

5.3.1. Create New RowPanel Class

Basic procedure to create a RowPanel instance is similar to the MainFrame.

1. Create a new CDMPanel instance
   - Syntax :

```
kwds['style'] = wx.SIMPLE_BORDER | wx.TAB_TRAVERSAL
cdml.CDMPanel.__init__(self, is_row = True, *args,
**kwds)
```

   - The second argument(is_row) should always be True.

2. Create temporary variables to save the data for a StudyModel object
   - Syntax : self.record = `cdml.GetInstanceAttr( 'DB class name' )`
   - Field variables are created and initial values defined in that class are assigned to the variables

2.1. Add controls to display/edit field value

3. Rules to add the controls are similar to those of MainFrame class.

3.1. Set Panel/Controls properties and positions

4. Disable the delete('x') button
   - If an object is given, delete button should be disabled
5. Set the data type of a control

- Set the same data type as the type of data field in a data object
- Default type is the ID_TYPE_ALPHA (i.e. string). Constants for the data types are shown in the Table below

Table. Data type constants for controls

Constant Name	Data Type
ID_TYPE_NONE	None[1)]
ID_TYPE_ALPHA	String. Default type
ID_TYPE_INTEG	Integer
ID_TYPE_FLOAT	Floating point
ID_TYPE_COMBO	Used only for combo control[2)]

1) Same as ID_TYPE_ALPHA. Implemented in one of previous version, but not used in current version.
2) If this type is set for a combo control, GUI uses the value in the text area to find a valid ID from the list control which is child of the popup window of the combo control.

6. The concept of the control positioning is same as the MainFrame class. Thus, detail descriptions are not presented here.

5.3.2. Bind Event Handlers

Most event handlers for controls in RowPanel class are bound to the controls when an RowPanel instance is created. In this section, steps to bind special event handler(s) to a control are described .

```
1:    self.cc_main_proc.GetTextCtrl().Bind(wx.EVT_LEFT_DCLICK,
self.OnLeftDblClick)
2:
3:    self.timer = wx.Timer(self, -1)
4:    self.Bind(wx.EVT_TIMER, self.OnTimer, self.timer)
```

Binding event handler

Above codes show how to add special event handler for a control. In line 1, the text control of cc_main_proc is bound to a event handler `OnLeftDblClick`. When user clicks twice that area ‘State’ form will be opened. In this case, focus checking isn’t necessary. Thus, dedicated event handler need to be implemented instead of the DefaultEventHandler. In future, this mechanism may be changed.
Line 3 and 4 show other example for timer control. Actually, these lines don’t work in current version. Same as line 1, this will be changed.

5.3.3. Implement Mandatory/Additional Methods

This section describes major difference between the MainFrame class. The primary function of the RowPanel class is to connect data to the database. Therefore, several methods should be implemented in each RowPanel class for data management. List of mandatory methods are shown in the Table below.

Table. List of mandatory methods for a RowPanel instance

Name	Function
GetValues	Gather data of each control
SetValues	Write data in each control
SaveValues	Gather data from each control, then save it to temporary variables
SaveRecord	Save data to the database object. If current RowPanel is new, create a new data object, then save it. If current RowPanel is old one, modify the data of database object
SetComboItem	Create item list for a combo control. All RowPanel instances that have combo control should implement this method. If there is no combo control in RowPanel instance, this method should not be implemented.

GetValues() method

- Syntax : GetValue(self):
- Create temporary variable
    - Syntax : record = copy.copy(self.record) ← need to import 'copy' module
    - Simply copy the record property of the RowPanel class
    - self.record property is created when new RowPanel instance is created.
    - For each GetValue method, the datatType property for each control is used. So, simiply call the GetValue method regardless of the type of the control.

SetValues() method

- Syntax : SetVaues(self, record, init=False):
    - record : database object related to this RowPanel instance
    - init : a parameter that can change the way data is set and is usually set to false. Depending on needs this option can be expanded or deprecated in the future.
- Basically, SetValues method is same as the inverse of the GetValues.
- Need to separate display methods
    - To decrease opening time of a form, different writing method has been implemented
    - When this method is called by Initialize method, all data are written as string type (If clause).
    - When it is called by CheckFocus method, SetValue() method is used. (Else clause)

SaveRecord() method

- Syntax : SaveRecord(self, record):

- record : temporary structure that contain value of controls.
- This method is a connection between the GUI and database. General sequence is
  1) Create a new database object by calling appropriate method in module 'DataDef' with relevant parameters.
  2) If current RowPanel instance is new, call AddNew() method defined in DataDef
  3) Else call Modify() method
- This method has to return a new database object to refresh the value of the controls.

SetComboItem() method

- Syntax : SetComboItem (self, record):
- Only RowPanel classes that contain combo control(s) should implement this method
- Create items list for a combo control
- Attach those list to combo controls using SetItems() method

6. Parameter Estimation (Lemonade Method)

6.1. Estimation Module:

This section describes the Estimation module that is a future implementation in the Python version and was already implemented in the Matlab Prototype. The estimation method is referred to as the **Lemonade Method** since "when study data give you lemons, make lemonade". Since this is a future implementation, some issues may change in the Python version. This section mainly describes the algorithms.

6.2. Expressions in Estimation Project:

In an estimation project, expressions will be evaluated in order to calculate the likelihood. The expressions will be built by the system to create a single expression that will be numerically evaluated repeatedly during optimization. Building the expression means substituting its token with their values until only leaf tokens exist in the expression. Coefficients will be represented as symbolic parameters while constructing the Likelihood expression.

6.3. Supporting Procedures and Functions

The following are important procedure and functions required to support the building of the Likelihood construction algorithm

6.3.1. Verification of Study/model/population correlation in Project:

Before parameters are estimated using the optimization process, the compatibility of a study, a model and the population set are checked. The following algorithm is used to check this, whenever a new model/study are added to a project or modified:

1. Reset the project status to have sufficient information for estimation $SufficientInfo = True$
2. Traverse the studies in the project using the index s and for each study $Studies_s$ in the project perform the following:
   2.1. Reset the used dimension list that will later be populated (all non coefficient parameters). $\mathbf{D} = \phi$
   2.2. Traverse all the transitions in the study $t_{ij}^{(s)}$ and all model transitions t_{ij} :

$$t \in \begin{Bmatrix} t_{ij}^{(s)} \mid t_{ij}^{s} \in Studies_s.Transitions \\ t_{ij} \mid t_{ij} \in Model.Transitions \end{Bmatrix}:$$

2.2.1.1.If the transition t is defined by categorical data:
2.2.1.1.1. For each dimension in the table except 'time'
$d \in t.Dimensions \wedge d \neq 'time'$
2.2.1.1.1.1.if this dimension was already encountered, i.e. if $d \in \mathbf{D}$, then:
2.2.1.1.1.1.1. If bounds exist, i.e. if $\mathbf{D}.Bounds_d \neq \phi$ then:
2.2.1.1.1.1.1.1.if Outermost Bound Ranges do not agree, i.e. if

$$\begin{pmatrix} \min(d.Bounds) \neq \min(\mathbf{D}.Bounds_d) \vee \\ \max(d.Bounds) \neq \max(\mathbf{D}.Bounds_d) \end{pmatrix}:$$

2.2.1.1.1.1.1.1.1. Raise an error indicating that this transition does not agree with previous bounds for this dimension
2.2.1.1.1.1.2. Else, if bounds do not exist, then:
2.2.1.1.1.1.2.1. $\mathbf{D}.Bounds_d = d.Bounds$
2.2.1.1.1.2.else, if the dimension was not on the used dimension list
$\mathbf{D} = \mathbf{D} \cup d$
2.2.1.2.Else If the transition t is from a study with a regression parameter vector $\mathbf{a}$ then:
2.2.1.2.1. For each dimension a_i in $\mathbf{a}$ perform the following:

2.2.1.2.1.1.Add the dimension to the used dimension list if not already there. $\mathbf{D} = \mathbf{D} \cup a_i$

2.2.1.3.Else if the transition t holds an expression, then:

2.2.1.3.1. Raise an error if non coefficient tokens a_i are encountered in $t.Probability$. Note that this may be handled in data entry so as assertion error will suffice.

2.3. Remove the dimension 'Time' from the collected data, i.e. $\mathbf{D} = \mathbf{D} - '\mathit{Time}'$

2.4. Check that all collected dimensions exist in population set $Population_s$

2.4.1. If $\mathbf{D} \not\subset Population_s.Dimensions$ than:

2.4.1.1.Indicate to the user that the population set is not suitable for this study and model combination by creating a string with all the missing dimensions. i.e. $Population_s.Dimensions - \mathbf{D}$. Another possibility is coloring line s in the GUI differently to indicate incompatibility.

2.4.1.2.Mark the project as not suitable to be estimated $SufficientInfo = False$

2.4.2. else if the dimension exists, perform the following:

2.4.2.1.for each dimension $d \in \mathbf{D}$ perform the following:

2.4.2.1.1. If bounds exist, i.e. if $\mathbf{D}.Bounds_d \neq \phi$ then:

2.4.2.1.1.1.verify that the population distribution is within the recorded dimension range, i.e. if

$$\begin{pmatrix} \min(Population_s.Dimensions_d.Bounds) < \min(\mathbf{D}.Bounds_d) \vee \\ \max(Population_s.Dimensions_d.Bounds) > \max(\mathbf{D}.Bounds_d) \end{pmatrix}$$

:

2.4.2.2. $\mathbf{D} \not\subset Population_s.Dimensions$:

6.3.2. Mean Calculation for Substitution from a Population Set:

Name: $v = \text{MeanCalcPop}(a, r, Population_s)$

Input:

- Dimension parameter: a
- Range parameter: $r = (r_{Low}, r_{High}]$. Note that the range is partially inclusive. The range is extracted from a table or is $r = (-\infty, \infty]$.
- Population Set: $Populations_s$

Output:

- Value parameter v holding the mean of the dimension parameter between the given ranges in the given population set.

Algorithm:

1. If the population set is defined as data:
   1.1. Calculate the average value v in the population of dimension a within the range specified by the given bounds within the given population set by:
      1.1.1. Reset the counter $n = 0$ and the sum $x = 0$
      1.1.2. If $r_{Low} = NaN$, meaning it's the first cell in an indexed table, then:
         1.1.2.1. Replace it with minus infinity, i.e. $r_{Low} = -\infty$
      1.1.3. Loop through all individuals j in the population set performing the following:
         1.1.3.1. If individual j has a value for dimension d, i.e. $Populations_s(j, a) \neq \phi$ then:
            1.1.3.1.1. If the value specified for individual j value for dimension d satisfies the given range, i.e. $Populations_s(j, a) \in r$ meaning that $r_{Low} < Populations_s(j, a) \leq r_{High}$ then:
               1.1.3.1.1.1. Increase the count by 1, i.e. $n \leftarrow n + 1$.
               1.1.3.1.1.2. Accumulate the value in the sum, i.e. $x \leftarrow x + Populations_s(j, a)$
      1.1.4. If $n = 0$ then Stop the procedure and report that the population selected for this study does not contain sufficient information for this dimension range. Consult the user to replace it with a categorical population with sufficient information that satisfies both model and study ranges and dimensions, or alternatively replacing the population with a population defined by a distribution.
      1.1.5. Calculate the average, i.e. $v = x / n$. Note that an invalid value will be returned in case of zero population.

2. If the population is defined as distribution functions:
   2.1. If the distribution for a is $Bernoulli(p)$ then:
      2.1.1. If $r = (-\infty, \infty]$ then $v = p$, meaning that the entire range is used.
      2.1.2. Otherwise $v = r_{High}$. Make sure the result is either 1 or 0 for consistency.
   2.2. Else if the distribution for a is $Binomial(n, p)$ then:
   2.3. v_i is extracted from the following equation:
      2.3.1. If $r = (-\infty, \infty]$ then $v = np$
      2.3.2. Otherwise
$$v = \frac{\sum_{i=r_{Low}}^{r_{High}} i\binom{n}{i}p^i(1-p)^{n-i}}{\sum_{i=r_{Low}}^{r_{High}}\binom{n}{i}p^i(1-p)^{n-i}} = np\frac{\sum_{i=r_{Low}-1}^{r_{High}-1}\binom{n-1}{i}p^i(1-p)^{n-i}}{\sum_{i=r_{Low}}^{r_{High}}\binom{n}{i}p^i(1-p)^{n-i}}$$
$$= np\frac{CDF(n-1, p, r_{High}-1) - CDF(n-1, p, r_{Low}-2)}{CDF(n, p, r_{High}) - CDF(n, p, r_{Low}-1)}$$
   2.4. Else if the distribution for a is $Geometric(p)$ then:
      2.4.1. If $r = (-\infty, \infty]$ then $v = 1/p$
      2.4.2. $v = \frac{-(r_{High}p+1)(1-p)^{r_{High}} + (r_{Low}p+1-p)(1-p)^{r_{Low}-1}}{p((1-p)^{r_{Low}-1} - (1-p)^{r_{High}})}$ Since the geometric distributions can be defined in two different ways. The definition that uses the # of draws including the first success was selected. In this definition, the expected value (mean) is: $E(x) = 1/p$.
   2.5. Else if the distribution for a is $Uniform(Low, High)$ then
$$v = \frac{\max(r_{Low}, Low) + \min(r_{high}, High)}{2}$$
   2.6. Else if the distribution for a is $normal(\mu, \sigma)$ then v is extracted from the following equation:
$$v^* = -\frac{\frac{1}{\sqrt{2\pi}}e^{-\frac{z_{High}^2}{2}} - \frac{1}{\sqrt{2\pi}}e^{-\frac{z_{Low}^2}{2}}}{\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{z_{High}} e^{-\frac{x^2}{2}}dx - \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{z_{Low}} e^{-\frac{x^2}{2}}dx}$$ and
$$= -\frac{PDF(z_{High}) - PDF(z_{Low})}{CDF(z_{High}) - CDF(z_{Low})}$$
$$z_{High} = \frac{(r_{High} - \mu)}{\sigma}, z_{Low} = \frac{(r_{Low} - \mu)}{\sigma}, v = \sigma v^* + \mu$$
   2.7. Else if the distribution for a is $Multinomial(n, m, p_1, p_2 ..., p_m)$ then $v = r_{High}$.
3. Return v

6.3.3. Calculate Prevalence from a Population Set:

Name: $w = \text{PrevalenceCalcPop}(\mathbf{D}, \mathbf{r}, Population_s)$

Input:

- A set of Dimensions: $\mathbf{D}$. These can be accessed with index $d \in \mathbf{D}$
- Range parameter for each dimension: $\mathbf{r} = \{r_1, .., r_d, .., r_{MaxDim}\}, r_d = [r_{d_{Low}}, r_{d_{High}}]$. Note that the range is inclusive for each member of the set, and that each member is indexed by the dimension index $d \in \mathbf{D}$
- Population Set: $Populations_s$

Output:

- The prevalence of individuals w in the given population that correspond to the given ranges in the given dimensions.

Algorithm:

1. If the population is given by a table:
   1.1. Reset the counter $v = 0$
   1.2. Loop through all people j in the population set
      1.2.1. Set a flag to true, i.e. $Flag = 1$
      1.2.2. For each dimension $d \in \mathbf{D}$ perform the following:
         1.2.2.1.If $r_{d_{Low}} = NaN$, meaning it's the first cell in an indexed table, then:
            1.2.2.1.1. Replace it with minus infinity, i.e. $r_{d_{Low}} = -\infty$
         1.2.2.2.If individual j has a value for dimension d and this is not a dummy dimension, i.e. $d \neq 'Dummy' \wedge Populations_s(j,a) \neq \phi$ then:
            1.2.2.2.1. If the value specified for individual j for dimension d does not satisfy the given range, i.e. $Populations_s(j,a) \notin r_d$ meaning that $\sim \left(r_{d_{Low}} < Populations_s(j,a) \leq r_{d_{High}} \right)$ then:
               1.2.2.2.1.1. $Flag = 0$
               1.2.2.2.1.2.Exit the dimension loop
      1.2.3. If $Flag = 1$ then
         1.2.3.1.Increase the count by 1, i.e. $v \leftarrow v + 1$.
   1.3. Divide the counter by the size of the population set $w \leftarrow v / Populations_s.Size$

2. Else if the population is defined by a distribution function:
   2.1. Reset the prevalence value $w = 1$
      2.1.1. For each dimension $d \in \mathbf{D}$ perform the following:
         2.1.1.1.If dimension $d \neq 'Dummy'$ then:

2.1.1.1.1. $w \leftarrow w * \left(Populations_s(d).CDF(r_{d_{High}}) - Populations_s(d).CDF(r_{d_{Low}}) \right)$

. Note that this will construct the following equation:

$$w = \prod_{d \in \mathbf{D}} \left(Populations_s(d).CDF(r_{d_{High}}) - Populations_s(d).CDF(r_{d_{Low}}) \right)$$

. Note that to support indexed table, the cumulative distribution function will return zero if the input is invalid, i.e. $CDF(NaN) = 0$.

3. Return *w*

6.3.4. Parse Markov Probability Term and Substitute Transition Parameters:

Name: $\{y^*, \mathbf{T}\} = \text{ParseMarkovTerm}(y, \mathbf{Index})$

Input:

- A symbolic probability expression y containing names model transitions names.
- A set of indices $\mathbf{Index} = (Index_1, Index_2 ..., Index_{MaxDim})$ that will allow accessing a table with dimensions $d \in \mathbf{D}$ associated with the model transitions.

Output:

- A set of transition names $\mathbf{T}$ that were contained in the expression
- The substituted probability y^*. This will represent the input probability expression with model terms substituted by data corresponding to the model transition table. If $\mathbf{Index} = \phi$, then no substitution will occur and the input will be returned after reconstruction, to help in debugging.

Algorithm:

1. Reset the string position indicator to the start $Pos = Start$.
2. Reset the output string to an empty string $y^* =$""
3. Reset the set of relevant model transitions $\mathbf{T} = \phi$
4. Until the string position indicator reaches the end of the string replace the tokens representing transition names by following these steps:
   4.1. Collect the next substring token A. The token is an uninterrupted sequence of alphanumeric characters, or a sequence of delimiting characters.
   4.2. If A is not a mathematical operation, not a function, not parenthesis, not a number, nor a prevalence, i.e. if $A \notin Function \wedge A \notin \{'*','+','-','^','(',')'\} \notin R \wedge A \notin \Re \wedge A \notin Prevalence$ then:
      4.2.1. Add the token to the relevant transition list $\mathbf{T} \leftarrow \mathbf{T} \cup A$ as it represents the transition parameter. Note that depending on the implementation language, the token A may be exchanged directly with the transition variable as the relation between the system reserved parameter and the transition is unique.
      4.2.2. If $\mathbf{Index} = \phi$, meaning no substitution is required, then:
         4.2.2.1. Just add the token to the output string: $y^* \leftarrow y^* + A$
      4.2.3. Else this means token substitution. Therefore perform the following:
         4.2.3.1. Replace the token A in the string with the value in the table of the model transition as indicated by the given indices. , i.e. $y^* \leftarrow y^* + (A.Data(Index_1, Index_2 ..., Index_{MaxDim}).Value)$. Note that the new value is enclosed in parenthesis to avoid operator priority conflicts, except when using function names and terminal tokens, or when already enclosed.
   4.3. Otherwise since the token will not be substituted, perform the following

4.3.1.1.1. Just add the token to the output string: $y^* \leftarrow y^* + A$

4.4. Advance the string indicator beyond the replaced token to be ready for the next token. $Pos \leftarrow Pos + A.Length$

4.5. Return $\{y^*, \mathbf{T}\}$

6.4. Likelihood Expression Construction Algorithm:

Following project verification the likelihood function will be constructed. Parameters will be estimated by constructing a likelihood expression by multiplying a likelihood expressions calculated for each study.

$$L(Coefficients) = \prod_{k=1}^{n} L_k(Coefficients)$$

This general likelihood function will later be optimized to extract the coefficient values. To achieve this, each study likelihood expression should be calculated by correlating the model, the study, and the population information provided in the project.

Here are some restrictions/prerequisites that are required for this algorithm to work properly:

1. Each study can represent states and transitions from only one sub-process in the model.
2. All transitions in the study must have one beginning state. This means that information on the starting number of people in each study is provided only once even in the case of multiple outcomes. This information is provided in a transition defined by starting from the beginning state and ending in NULL.
3. Currently, all information in transitions in the study must have the same dimensions and ranges, except for Time. This means Study dimensions are unified already.
4. Information for the study is provided either categorically or by regression parameters. Different transitions in the same study will not mix these types of information.
5. Study transitions that are defined but do not contain any data/probability information signify a retrospective study. These transitions are zeroed out of the theoretical model by the algorithm.
6. A model will not contain loops among consecutive event states or involving event states.
7. Each study is fully contained within a single sub-process and the traversed by the study cannot “skip” or represent a sub-process contained within the sub-process it describes.
8. Categorical study data is represented in the transitions such that all dimensions within the study are consistent:
    a. Output Times for all transitions is the same and the last time corresponds to the study length.
    b. Ruling out time, all table dimensions are the same and having the same ranges
    c. The transition from the beginning state to null holds the initial number of individuals in the population. It must be defined and the table dimensions and ranges are the same as all other transitions.

   d. Constant numbers entered by the user as transitions are automatically converted by the system to a table with the time dimension and a single cell.
9. Transitions that signify staying in the same states are not allowed as user input. Rather, they are calculated within the system.
10. Population information is defined either by data or by defining marginal distributions for each dimension separately. Joint distributions are not used.
11. It is assumed that the population information defined by data is dense enough to support all the combinations of the dimension ranges. No interpolation or extrapolation of population data is performed.
12. Regression studies are limited by the form of functions that they can handle. Currently, linear, exponential, and UKPDS are used. Moreover, the data is supplied as a coefficient vector and a parameter vector. Each member of the parameter vector can be either a constant or an expression which is a function of a single parameter.
13. Dimension Ranges for the same dimension in the model and the study should have the same maximum and minimum values. In between range values may vary, however extreme bounds must agree. In other words, range interpolation is supported while extrapolation is not supported.
14. Multinomial distribution is not fully supported.
15. Nested sub-processes in the model are not supported since transitions into a state within the nested sub-process is not resolved by the system. Nevertheless an outer loop that distinguished sub-processes exists so it may be possible in some simple cases to estimate parameters in multiple sub-processes together.
16. Currently only one event state is supported as a study outcome. Also mixing event states and non event states as outcomes of the same study is not supported.

6.4.1. Notes Regarding the Algorithm

Since transitions in study and in model can be defined differently, either categorically or by regression information, or with different dimensions to be considered, the algorithm that builds the likelihood expression has to bridge the gap between these differences. The proposed algorithm deals with that by preprocessing the study and the model transitions and reorganizing the information in a common/unified representation.

The common/unified representation is a multidimensional table combining dimensions used in the study and the model. The range bounds for each dimension include the range bounds from each transition in the study and the model.

After preprocessing, each dimension in the transition table in a model will be of the same size. After preprocessing for studies with categorical data, each cell in a transition table in the study holds an observed count of people. Since the table is of the same size for all transitions, this allows correlation between study and model under the same conditions.

The table below shows how differences between study and model are handled during preprocessing and likelihood construction.

	Model transition is an expression (regression)	Model transition is a table (categorical)	Model transition is a single coefficient (scalar)
Study provides regression data	**Currently unsupported**	**Preprocessing:** **Model** and **Study** remain unchanged. **Likelihood:** The **Study** data is used to the translated likelihood expression. **Correlation:** The **Study** and the **Model** data are combined with population information to construct the correlation term. While assigning population data to the correct table cell.	Treated as an expression.
Study provides categorical data in a table	**Currently unsupported** Currently only simple expressions that do not involve covariates are supported.	**Preprocessing:** **Model** and **Study** tables is collapsed/expanded to fit the unified dimensions. **Likelihood:** Unified table cells are traversed to construct likelihood terms. **Correlation:** None	Treated as categorical data with a single cell table in the model.
Study provides a constant probability	Currently only simple expressions that do not involve covariates are supported. Treated as categorical data with a single cell table in the study	Treated as categorical data with a single cell table in the study	Treated as categorical data with a single cell table in the study and in the model.

6.4.2. Algorithm: Construct and Optimize Likelihood Expression:

Definition:

$(L, \overline{Coefficients}) = \text{ConstructAndOptimizeLikelihood}(Model, \mathbf{Studies}, \mathbf{Populations})$

Input:

- The Model in the project: *Model*
- The set of Studies contained in the project: $\mathbf{Studies} = \{Studies_s\}$
- The set of populations contained in the project and correspond to the given studies: $\mathbf{Populations} = \{Populations_s\}$. Note that an empty population is possible, signifying that a study does not have a population associated with it. However, in any case each study must have a population variable associated with it and therefore the sizes of these vectors are the same, i.e.: $\|\mathbf{Populations}\| = \|\mathbf{Studies}\|$.
- Several system constants to control numerical calculations such as convergence tolerances, derivation intervals etc.

Output:

- The likelihood L associated with the project that takes into account the model, all the studies and their associated population sets as a function of coefficient parameters supplied by the user as part of the model. The likelihood is a function of model coefficients variables $L(Coefficients)$
- A set of values $\overline{Coefficients}$ corresponding to all coefficients used in the model transitions. These represent estimated probability components of the model after optimization of the likelihood.

Algorithm:

1. Reset the general Log likelihood expression, $L = 0$.
2. For each sub process in the model

2.1. Construct the model matrix $\mathbf{P}^{**}$ using the following steps:

Initially Construct **Model** Probability Matrix

2.1.1. Assign a consecutive number for each state in the sub process that will indicate an index to a matrix: $i = 1..n$, such that $SubProcess.States_i = Model.States_{StateID} \in SubProcess$, where $StateID$ is the unique ID of the state while i is an index of this state in the sub-process list.

2.1.2. Init the non event probability matrix $\mathbf{R} = [0]_{nxn}$ where n is the number of states in the sub process.

2.1.3. Init the event probability matrix to a unity matrix, $\mathbf{E} = \mathbf{I}_{nxn}$ where n is the number of states in the sub process.

2.1.4. Reset the event state counter $c = 0$

2.1.5. For each transition in the model sub-process between states t_{ij}:

2.1.5.1.If i is not an event state, i.e. $SubProcess.States_i.Event = False$ then:

2.1.5.1.1. Populate the non event probability matrix element associated with the transition with the name of the transition, i.e. $r_{ij} = t_{ij}$. Note that this is not the same as using the parameter name in the transition probability as two different transitions may use the same parameter. Therefore the name of the transition is generated as a temporary system reserved parameter and is assigned with the probability parameter given by transition probability function.

2.1.5.2.Otherwise , meaning that an event state is processed and therefore: $SubProcess.States_i.Event = True$ perform the following:

2.1.5.2.1. Increase the event state counter $c \leftarrow c + 1$

2.1.5.2.2. Reset the diagonal element in the event probability matrix associated with the originating event state, i.e. $e_{ii} = 0$

2.1.5.2.3. Populate the event probability matrix element associated with the transition with the name of the transition, i.e. $e_{ij} = t_{ij}$.

2.1.5.3.Complete the construction of the non event probability matrix element. Populate the diagonal elements of the matrix with a complementary value that will sum the row members to 1: $r_{ii} = 1 - \sum_{j=1}^{n} r_{ij}$.

2.2. For each study with index $s = 1..|Studies|$:

Loop through **Studies**

2.2.1. If not all the states in the study or in its pooled states are part of the sub-process, exit this loop and continue to the next study. In other words, if the study is not in this sub-process, ignore it.

2.2.2. If the study is already marked as used raise an error, otherwise mark the study as used. Note that the 'if' statement is an assertion check that checks the validity of the program and is not essential to the algorithm. However, the marking as used is important for the consistency check at the end of the algorithm.

2.2.3. If the study outcome states are all non event states, then build the components of the event including model matrix in the following manner:

2.2.3.1.Copy the modified event matrix from the event matrix. $\mathbf{A} = \mathbf{E}$

2.2.4. Else if the study outcome states include a single event state indexed as k in the matrix, then build the event ignoring matrix components:

2.2.4.1.Create a modified event matrix $\mathbf{A}$ by setting the row corresponding to the event state to have 1 on the diagonal and 0 elsewhere. i.e.

$$a_{ij} = \begin{cases} e_{ij} & i \neq k \\ 0 & i = k \wedge j \neq i \\ 1 & i = j = k \end{cases}$$

2.2.5. Compute the consecutive event probability matrix for the modified matrix, $\mathbf{A}^* = \mathbf{A}^c$ to account to a possible series of event states. The event state

counter c serves here as an upper bound of possible consecutive transitions. Generally, it is possible that the power raising will not change the probability. Note that loops are not allowed.

2.2.6. Verify that there were no loops in the modified consecutive event probability matrix $\mathbf{A}^*$, by checking the diagonal. If any $a^*_{ii} \neq 0 \wedge a_{ii} = 0$, then raise an error indicating a loop of event states exists.

2.2.7. Define the model matrix $\mathbf{P}$ as $\mathbf{P} = \mathbf{R} * \mathbf{A}^*$. Note that this matrix may not include any representation of event states that were previously sunk and that the diagonal will later be changed.

2.2.8. Otherwise, if more than one outcome event state is defined by the study, or if the study mixes event and non event states, then raise an error saying that multiple event state outcomes are not supported by a project.

2.2.9. Construct the $\mathbf{K}$ and $\hat{\mathbf{K}}$ matrices and modify the $\mathbf{P}$ matrix:

Take care of pooled states in **Study**

2.2.9.1. Init $\mathbf{K} = [0]_{(mxn)}$ and $\hat{\mathbf{K}} = [0]_{(nxm)}$ where m is the number of study states.

2.2.9.2. For each sub-process state $Model.State_i$ indexed by $i = 1...n$ perform the following:

2.2.9.2.1. If this state exists in the study as a regular state i.e. $Model.State_i = Studies_s.State_j$ indexed by $j = 1...m$ then:

2.2.9.2.1.1. $k_{ji} = 1$

2.2.9.2.1.2. $\hat{k}_{ij} = 1$

2.2.9.2.2. Else if this state exists in the study as a part of a pooled state $Studies_s.States_z$ i.e. $Model.State_i \in Studies_s.States_z = Pooled_z$ then:

Mark outcome states in the **Study** by making them sinks in the **Model** Probability matrix

2.2.9.2.2.1. $k_{zi} = w_{zi}$, where w_{zi} is the prevalence of state i in state z

2.2.9.2.2.2. $\hat{k}_{iz} = 1$

2.2.9.2.3. Traverse all transitions belonging to the study: $t^{(s)}_{hk} \in Studies_s$

2.2.9.2.3.1. If model state i is an outcome state in the study, i.e. $Model.State_i = Studies_s.State_k$, then make it a sink in $\mathbf{P}$ by:

2.2.9.2.3.1.1. Reset the row i in $\mathbf{P}$ to zero, while keeping the diagonal a unity: $p_{ij} = 0, \forall j = 1..n \wedge j \neq i, p_{ii} = 1$

Consistency check for prevalence values of pooled states

2.2.9.3. Perform a consistency check that the members of row i in matrix $\mathbf{K}$ sum to 1 by:

2.2.9.3.1. Traverse the rows $j = 1...m$ performing the following:

2.2.9.3.1.1. Resetting the sum $b = 0$

2.2.9.3.1.2. Traverse the columns $i = 1...n$ and for each column:

2.2.9.3.1.2.1. Add the matrix member to the sum $b \leftarrow b + k_{ij}$

2.2.9.3.1.3. If $b \neq 1$ raise an error to the user saying that prevalence values associated with pooled state $Studies_s.States_j$ do not sum to 1 and require user update.

Take care of retrospective **Studies**

2.2.9.4.Take care of the case of retrospective studies by traversing the transitions in the study, and for each transition $t_{hk}^{(s)} \in Studies_s$ perform the following:

2.2.9.4.1. If the transition has no data, either categorical or regression, i.e. $t_{hk}^{(s)}.Probability = \phi$ and $t_{jk}.RegressionParamVector = \phi$. This indicates a retrospective study transition and therefore:

2.2.9.4.1.1.If $h = NULL$, meaning all transitions to state k, then zero out the entire column of $\mathbf{P}$, while keeping the diagonal a unity: $p_{jk} = 0, \forall j = 1..n \wedge j \neq k, p_{kk} = 1$. This should not be confused with a categorical start data transition, that is signified with $k = NULL$ and must have categorical data defined.

2.2.9.4.1.2.Else this means only a single transition requires resetting, i.e. $p_{hk} = 0$

Calculate **Model** and **Study** timed probability matrices

2.2.10. The previous operations created changes in the matrix sum and therefore the diagonal needs modification. This is performed by traversing all indices $i = 1..n$ and performing the following:

2.2.10.1. Populate the diagonal elements of the matrix with a complementary value that will sum the row members to 1:

$p_{ii} = 1 - \sum_{j=1}^{n} p_{ij}$. Note that the matrix contains model transition parameter names.

2.2.11. For each year of the study until its maximum time $\tau = 1..Sudies_s.Time$ calculate the corresponding transition matrices:

2.2.11.1. Calculate time probability: $\mathbf{P}^{\tau} = \mathbf{P}^{\tau-1}\mathbf{P}$, where $\mathbf{P}^{0} = \mathbf{I}$

2.2.11.2. Calculate grouped nodes probability: $\mathbf{Y}^{\tau} = \mathbf{K}\mathbf{P}^{\tau}\hat{\mathbf{K}}$

Find all relevant **Model** transitions with respect to the **Study**

2.2.12. Find all model transitions that are relevant to the study by performing the following:

2.2.12.1. Reset the set of relevant model transitions $\mathbf{T} = \phi$

2.2.12.2. Traverse all the transitions $t_{ij}^{(s)}$ in the study $t_{ij}^{(s)} \in Studies_s.Transitions$ and for each transition performing the following:

2.2.12.2.1. If $i \neq NULL \wedge j \neq NULL$ then perform the following:

2.2.12.2.1.1. Parse the symbolic expression string contained by y_{ij}^{τ} in the probability matrix $\mathbf{Y}^{\tau}$ for time $\tau = Studies_s.Time$ and collect all tokens representing transition names by calling the following function: $\{y^*, \mathbf{T}^*\} = \text{ParseMarkovTerm}(y_{ij}^{\tau}, \phi)$. The output model transitions $\mathbf{T}^*$ are associated with study transition $t_{ij}^{(s)}$.

Calculate the unified **Study**-**Model** table dimension set and range set

2.2.12.2.1.2. Combine the output to the set of model transition, i.e. $\mathbf{T} \leftarrow \mathbf{T} \cup \mathbf{T}^*$

2.2.13. Reset the used dimension list that will later be populated (all non coefficient parameters). $\mathbf{D} = \phi$

2.2.14. Reset the study dimension list that will later be populated (all non coefficient parameters). $\mathbf{D}_s = \phi$

2.2.15. Determine the set of Transitions by which the unified dimension set for this study and model combination. The set of transitions to be used $\mathbf{T}^{**}$ is a combination of study and the model transitions, i.e. $\mathbf{T}^{**} = t_{ij}^{(s)} \cup \mathbf{T}$. Traverse the study transitions first.

2.2.16. Traverse all study transitions $t \in t_{ij}^{(s)}$:

2.2.17. If the transition t has categorical data then:

2.2.17.1. If this is the first study transition,

2.2.17.1.1. Define the Study dimensions as the transition dimensions sans the time dimension, i.e. $\mathbf{D}_s = t.Dimensions - '\mathit{time}'$. Note that bounds are also copied. Subsequent changes to bounds will not be reflected in $\mathbf{D}_s$. In any case, make sure that $\mathbf{D}_s \neq \phi$ by introducing a 'Dummy' dimension if needed.

2.2.17.2. Otherwise, meaning that $\mathbf{D}_s \neq \phi$ i.e. this is a study transition and it is not the first.

2.2.17.2.1. Verify that $t.Dimensions - '\mathit{Time}' = \mathbf{D}_s$ and that all bounds are the same and raise an error if not. This is done to ensure that transitions with different dimensions within the study will not appear as this is not supported.

2.2.18. If $\mathbf{D}_s = \phi$, meaning all variables are constants then:

2.2.18.1. Create a new dummy dimension $\mathbf{D}_s = \{'\mathit{dummy}'\}$ with the range $\mathit{Bounds}_{\mathit{dummy}} = \{-\infty, \infty\}$. This line is here to allow consistency later on as the continuation assumes there is at least one table dimension with one cell. Depending on implementation method, this may not be required if constants and tables with one cell can be easily associated.

2.2.19. Initialize the unified dimension list to be the same as the study dimension list, i.e. $\mathbf{D} = \mathbf{D}_s$. Note that the face that the study dimensions are defined first in $\mathbf{D}$ is important and will later be used to map cells in $\mathbf{D}$ to $\mathbf{D}_s$, It is important to keep this relation when traversing the tables later on.

2.2.20. Traverse all relevant model transitions $t \in \mathbf{T}$:

2.2.20.1.1. For each dimension in the table $d \in t.Dimensions$ perform the following:

2.2.20.1.1.1. Just make an assertion check that the dimension is not 'time' and raise an error is it is as time should not be present in the model.

2.2.20.1.1.2. Add the dimension to the used dimension list if not already there, i.e. $\mathbf{D} \leftarrow \mathbf{D} \cup d$.

2.2.20.1.1.2.1. Add the range bounding values specified for this dimension in the study/model to the dimension range list, while ignoring multiple values of range bounds. $\mathbf{D}.Bounds_d = \mathbf{D}.Bounds_d \cup t.Dimentions_d.\{Bounds\}$ Raise an error in case of range incompatibility for a dimension arising from trying to combine tables with range values and tables with indices. Consider the range as a sorted list in ascending order.

2.2.21. If $\mathbf{D} = \phi$, meaning all variables are constants then:

2.2.21.1. Create a new dummy dimension $\mathbf{D} = \{'dummy'\}$ with the range $Bounds_{dummy} = \{-\infty, \infty\}$. This line is here to allow consistency later on as the continuation assumes there is at least one table dimension with one cell. Depending on implementation method, this may not be required if constants and tables with one cell can be easily associated.

Model Preprocessing

2.2.22. Again traverse all the transitions in the **model** related to the study $t \in \mathbf{T}$

2.2.22.1. Create a multidimensional table $t.Data$ in the transition definition, possibly overriding a pervious definition. Set the data of the transition to have the same dimension as the model, i.e. $t.Data = \mathbf{D}$.

Convert **Model** coefficients to the common table form

2.2.22.2. If the transition formula $t.Probability$ holds a single coefficient β:

2.2.22.2.1. Traverse all cells by recursively traversing all dimensions of the table to set them to the coefficient value $t.Data(1..Dim_1.MaxIndex,1..Dim_2.MaxIndex,...,1..Dim_{MaxDim}.MaxIndex) = \beta$

Convert **Model** expressions to the common table form

2.2.22.3. If the model contains a transition formula and the study $Studies_s$ is categorical, it may hold an expression combined from coefficients and parameters, in which case mean substitution may be required and therefore perform the following:

2.2.22.3.1. Parse the expression string to extract all parameters using the following function: $\{y^*, \mathbf{T}\} = \text{ParseMarkovTerm}(t.Probability, \phi)$.

2.2.22.3.2. Reset the non coefficient parameter vector to $\mathbf{a} = \phi$

2.2.22.3.3. Form the non coefficient parameter vector by traversing all dimensions and for each parameter $a_i \in \mathbf{T}$ perform the following:

2.2.22.3.3.1.1. If a_i is not a coefficient, then:

2.2.22.3.3.1.1.1. Add it to the list, i.e. $\mathbf{a} = \mathbf{a} \cup a_i$

2.2.22.3.4. Traverse all cells by recursively traversing all dimensions of the table $t.Data$ previously prepared for this transition. For each table cell $c = t.Data(Index_1, Index_2..., Index_{MaxDim})$ perform the following:

2.2.22.3.4.1. Reset the expression $y^* = t.Probability$

2.2.22.3.4.2. For each parameter $a_i \in \mathbf{a}$ perform the following:

2.2.22.3.4.2.1. Exchange each member a_i of **a** with a value v_i by traversing the vector of parameters with index i and performing the following:

2.2.22.3.4.2.1.1. If a_i is a constant then $v_i = a_i$

2.2.22.3.4.2.1.2. Else If a_i is defined in the table $a_i \in t.Data.Dimentions$ then perform the following:

2.2.22.3.4.2.1.2.1. Extract the ranges specified for this dimension in the cell:
$r_i = [r_{Low}, r_{High}] = [LowBound(c, a_i), UpperBound(c, a_i)]$
note that the functions will return the bounding range values in case of range tables and indices in case of index tables.

2.2.22.3.4.2.1.3. Else, meaning that a_i is not defined in the table and therefore:

2.2.22.3.4.2.1.3.1. Define the range as
$r_i = [r_{Low}, r_{High}] = [-\infty, \infty]$

2.2.22.3.4.2.1.4. Call the function mean calculation function with the current population set, parameter, and range $v_i = \text{MeanCalcPop}(a_i, r_i, Population_s)$

2.2.22.3.4.2.1.5. Substitute the parameter a_i in y^* with the mean v_i i.e. $y^* \leftarrow y^*(a_i \leftarrow v_i)$ where $y^* = t.Probability$

2.2.22.3.4.2.2. Place the expression y^* in the cell c in the table, i.e. $c.Value = y^*$

2.2.22.4. If the transition holds a table:

2.2.22.4.1. Create a source Table by copying the Probability table. i.e.:
$Source = t.Probability$

Convert the **Model** table to the common table form

2.2.22.4.2. Traverse all cells by recursively traversing all dimensions of the destination table $t.Data$ previously prepared for this transition. For each table cell $c = t.Data(Index_1, Index_2 ..., Index_{MaxDim})$ perform the following:

2.2.22.4.2.1. Traverse all dimensions in **D** and for each dimension $d \in \mathbf{D}$ perform the following:

2.2.22.4.2.1.1. Record the cell bounds from the destination table to allow source table lookup later:
$r_d = [r_{d_{Low}}, r_{d_{High}}] = [LowBound(c, a_i), UpperBound(c, a_i)]$

2.2.22.4.2.1.2. If the dimension d exists in the source table provided by the study transition formula, i.e. if $d \in Source.Dimensions$ then:

2.2.22.4.2.1.2.1. Locate the table index $Index_d^*$ that corresponds with dimension d in the source table. This index will satisfy:
$r_{d_{Low}}^* < r_{d_{High}} \le r_{d_{High}}^* \wedge r_{d_{Low}}^* \neq NaN \vee r_{d_{High}} \le r_{d_{High}}^* \wedge r_{d_{Low}}^* = NaN$
where $r_{d_{Low}}^* = Source.Bounds(d)_{index_d^*}$ and $r_{d_{High}}^* = Source.Bounds(d)_{index_d^*+1}$. Raise an error if such an index does not exist. Note that NaN will require specialized treatment in these computations to avoid using it in computations. One way to do this is replacing it with $-\infty$ and asking only if $r_{d_{Low}}^* < r_{d_{High}} \le r_{d_{High}}^*$.

2.2.22.4.2.2. Access the source table value with the set of collected indices that are relevant to the source table and extract the cell value:
$v = Source(Index_1^*, Index_2^* ..., Index_{MaxDim^*}^*).Value$ where $MaxDim^* = Source.Dimensions.MaxDim$ and the indices are ordered properly according to the dimension index as it is counted in the source table.

2.2.22.4.2.3. Set the destination table cell to this value:
$c.Value = v$

2.2.23. Again traverse all transitions $t \in Studies_s$ in the **study** and perform the following:

2.2.23.1. If the transition state indices are meaningful, and the transition does not hold regression data, i.e. if $i \neq NULL \wedge t.RegressionCoefVector = \phi$ then:

2.2.23.1.1. If the transition holds a constant, then:

2.2.23.1.1.1. Create a Source Table $Source$ with a Dummy dimension that has only a single Cell with the range $Source.Dimensions = \{'dummy'\}$ with the range $dummy.\{Bounds\} = \{-\infty, \infty\}$. Note that in this case, there can be a shortcut later while calculating the source prevalence. This conversion is done so that dealing will be done at the table cell level. Note that in this case $\mathbf{D}_s$ should have the dummy dimension only also.

2.2.23.1.1.2. Copy the constant in the probability to a single cell in the table $Source.(CellIndex) = t.Probability$

2.2.23.1.2. Else if the transition holds categorical data, then:

2.2.23.1.2.1. Create The source table by copying the entire table in the transition along with its dimension definitions: i.e. $Source = t.Probability$. Note that adding a Dummy

Study
Preprocessing:
Map **Study** table or constant to the common table form

dimension may be needed if only the dimension 'Time' exists.

2.2.23.1.3. Create the **mapping index table** that maps between cell ranges in $\mathbf{D}$ and a specific cell in $\mathbf{D}_s$. For this, construct an empty table the size of $\mathbf{D}_s$. Note that each cell in this table will hold a list of indices rather than a single number. This can be implemented in several ways during coding. One way is actually using a list structure, another is adding another temporary system dimension to the table and using it to store incoming numbers. In any case reset the mapping index table $t.\mathbf{MapIndexTable} = \phi_{|\mathbf{D}_s|}$.

2.2.23.1.4. Create the **mapping cell prevalence table** of size $\mathbf{D}$, i.e. $t.\mathbf{MapCellPrevalenceTable} = \phi_{|\mathbf{D}|}$. This table will retain the prevalence of model cells with respect to study cells.

2.2.23.1.5. Extract the Range of the 'Time' dimension from the source table ignoring the NaN at the beginning, i.e. $TimeRange = t.Source.Dimensions(\text{'}Time\text{'}).Range - NaN$.

2.2.23.1.6. If $TimeRange = \phi$, meaning time does not exist in the table,

2.2.23.1.6.1. If this is not a transition indicating initial numbers and a transition where the probability is defined as a table , i.e. $j \neq NULL$ and $t.Probability$ is a table then:

2.2.23.1.6.1.1. Raise an error indicating that a time dimension should have been defined explicitly in the table.

2.2.23.1.6.2. Else If the transition $t.Probability$ holds a constant or an expression:

2.2.23.1.6.2.1. Extract the Range of the 'Time' dimension from the study. $TimeRange = Studies_s.StudyLength$

2.2.23.1.6.3. Define a new dimension range that includes the 'Time' dimension: $\mathbf{D}_s^t = \mathbf{D}_s \cup \text{'}Time\text{'}$.

2.2.23.1.7. Else, meaning it is a transition indicating initial numbers then:

2.2.23.1.7.1. The new dimension range is defined the same way as the unified dimension: $\mathbf{D}_s^t = \mathbf{D}_s$

2.2.23.1.8. Create a new empty multidimensional table $t.Data$ with all the used dimensions in $\mathbf{D}_s^t$, possibly including the 'Time' dimension. Note that the range of the 'Time' dimension should be the same as the extracted data from the transition.

2.2.23.1.9. Record the time information in a separate vector for later use. For this remove the NaN at the beginning of the vector. i.e. $t.TimeSteps = TimeRange - NaN$

2.2.23.1.10. Create a dimension mapping index vector $\mathbf{a}$ between the source table and the $\mathbf{D}_s^t$ and initialize it with zeros. It should map

dimension indices only and its size is the same as the number of dimensions in $\mathbf{D}_s^t$.

2.2.23.1.11. Traverse all dimensions $d_{SourceIndex} \in Source.Dimensions$ in the source table and for each one :

2.2.23.1.11.1. Traverse all dimensions $d^*_{DestIndex} \in \mathbf{D}_s^t$ in the destination table and for each one :

2.2.23.1.11.1.1. If $d^*_{DestIndex} = d_{SourceIndex}$ then :

2.2.23.1.11.1.1.1. Update **a** such that $a_{DestIndex} = SourceIndex$. This will allow remapping the tables so that the time dimension will be defined at the end. In many cases, this vector may look like just a count upwards.

2.2.23.1.12. Traverse all cells by recursively traversing all dimensions of the destination table $t.Data$. For each table cell $c = t.Data(Index_1, Index_2 ..., Index_{MaxDim})$ perform the following:

2.2.23.1.12.1. Calculate the corresponding source table cell c^* by using the mapping index vector **a** . That is $c^* = Source.Data(Index_{a_1}, Index_{a_2} ..., Index_{a_{MAxDim}})$

2.2.23.1.12.2. Copy source cell to the destination cell. i.e. $c = c^*$. This feat essentially will reorder dimensions such that 'time', if appears, has the last index. This will greatly simplify matters later on.

2.2.23.2. Traverse all Model cells by recursively traversing all dimensions of $\mathbf{D}$, for each table cell index $\mathbf{Index} = (Index_1, Index_2 ..., Index_{MaxModelDim})$ perform the following:

2.2.23.2.1.1. Traverse all dimensions in $\mathbf{D}$ and for each dimension $d \in \mathbf{D}$ perform the following:

2.2.23.2.1.1.1. Record the cell bounds from the model table according to $\mathbf{D}$ to allow table lookup later, i.e. $r_d = [r_{d_{Low}}, r_{d_{High}}] = [LowBound(c, a_i), UpperBound(c, a_i)]$ And organize these in a vector $\mathbf{r} = \{r_1, .., r_d, .., r_{MaxDim}\}$

2.2.23.2.1.1.2. If the dimension d exists in the study table dimensions. This should happen only for the first few dimensions as study dimensions will appear first, i.e. if $d \in \mathbf{D}_s$ then:

2.2.23.2.1.1.2.1. Locate the table index $Index^*_d$ that corresponds with dimension d in the study table. This index will satisfy: $r^*_{d_{Low}} < r_{d_{High}} \le r^*_{d_{High}} \wedge r^*_{d_{Low}} \ne NaN \vee r_{d_{High}} \le r^*_{d_{High}} \wedge r^*_{d_{Low}} = NaN$

where $r^*_{d_{Low}} = \mathbf{D}_s.Bounds(d)_{index^*_d}$ and $r^*_{d_{High}} = \mathbf{D}_s.Bounds(d)_{index^*_d+1}$. Raise an error if such an index does not exist. Note that NaN will require specialized treatment in these computations to avoid using it in calculation. One way to do this is replacing it with $-\infty$ and asking only if $r^*_{d_{Low}} < r_{d_{High}} \le r^*_{d_{High}}$.

2.2.23.2.1.1.2.2. Organize the calculated ranges in a vector $\mathbf{r}^* = \left\{r^*_1,..,r^*_d,..,r^*_{MaxStudyDim}\right\}$

2.2.23.2.1.1.2.3. Organize the calculated indices $\mathbf{Index}^* = \left\{Index^*_1....,Index^*_d,Index^*_{MaxStudyDim}\right\}$.

2.2.23.2.1.2. Update the Mapping index table with the newly calculated index that ties the model cell index to the corresponding study cell index. i.e. $t.\mathbf{MapIndexTable}(\mathbf{Index}^*) \leftarrow \left\{t.\mathbf{MapIndexTable}(\mathbf{Index}^*),\mathbf{Index}\right\}$. Note that each cell will host a vector of multidimensional indices or a vector of flattened indices which may be a good way to implement this. There may be other ways to implement this.

2.2.23.2.1.3. Extract the prevalence value associated with the model table by calling the following function with destination table characteristics: $x = \text{PrevalenceCalcPop}(\mathbf{D},\mathbf{r},Population_s)$.

2.2.23.2.1.4. Extract the prevalence value associated with the study table by calling the following function with source table characteristics: $y = \text{PrevalenceCalcPop}(\mathbf{D}_s,\mathbf{r}^*,Population_s)$. Note that the 'Time' dimension does not take place in this calculation.

End of preprocessing, **Study** and **Model** are now compatible

2.2.23.2.1.5. Update the mapping cell prevalence table to the prevalence table in the appropriate index by dividing the model prevalence by the study prevalence $t.\mathbf{MapCellPrevalenceTable}(\mathbf{Index}) = x/y$

2.2.24. Reset the study Log likelihood expression, i.e. $L^* = 0$.

2.2.25. Reset the study correlation expression, i.e. $L^*_{correlation} = 0$.

2.2.26. If the study transitions are defined by regression transitions, then construct the study Log Likelihood expression L^* which will be composed of regression by traversing all transitions in the study and for each transition $t_{jk} \in Studies_s$ that is not a retrospective study transition, i.e. $t_{jk}.RegressionCoefVector \neq \phi$ perform the following:

Construct the **Regression Study Log Likelihood** term

2.2.26.1. Construct a system reserved temporary coefficient vector $\boldsymbol{\gamma}$ of length n^*. The value n^* is determined by the length of the regression coefficient vector defined in the transition i.e. $n^* = length(t_{jk}.RegressionCoefVector)$. Note that this is a new system reserved variable and in order to avoid name conflict the transition indices are used. Also note that these coefficients will later be used as functions of other parameters in the regression parameter vector and therefore some substitutions may be required.

2.2.26.2. Create the temporary model coefficient table *TempModelCoefTable* of dimensions **D**

2.2.26.3. Create a temporary individual count table *TempIndividualCount* of dimension **D** and reset all its cells to zero.

2.2.26.4. For each count $c = 0..2n^*$ that will later be used to index perturbations perform the following:

2.2.26.4.1. Create a temporary study term accumulator table $TempStudyTermAccTable_c$ each of dimension **D**. Note that there will be $2n^* + 1$ tables like that.

2.2.26.4.2. Reset all the cells of $TempStudyTermAccTable_c$ to zero

2.2.26.5. If $\mathbf{D} = \{'Dummy'\}$ then:

2.2.26.5.1. Create the symbolic probability coefficient parameter η

2.2.26.5.2. Populate the single table cell in *TempModelCoefTable* with η

2.2.26.6. Else if $\mathbf{D} = \{'Dummy'\}$ then:

2.2.26.6.1. Traverse all cells defined in **D** by recursively traversing all dimensions. For each table cell index set $\mathbf{Index} = (Index_1, Index_2 ..., Index_{MaxDim})$ perform the following:

2.2.26.6.1.1. Create the symbolic probability coefficient parameter $\eta_{Index_1\_Index_2\_...\_Index_{MaxDim}}$. Note that the indices represent a location in the table that will later be used. Also note that another method for pointing the table index can be used such as addressing the table as a long vector.

2.2.26.6.1.2. Populate the temporary model coefficient table cell with the newly created coefficient. i.e. $TempModelCoefTable(\mathbf{Index}) = \eta_{Index_1\_Index_2\_...\_Index_{MaxDim}}$

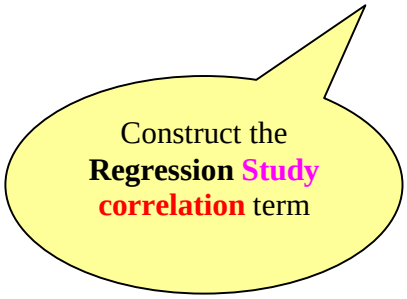
2.2.26.7. If no population set is defined for the study, then create a new population set with one dimension 'Dummy' with one record where it is set to zero.

2.2.26.8. If the population set for the Study is categorical then perform the following:

Regression Correlation Terms for Categorical Population Data

2.2.26.8.1. Reset the correlation term for the study transition $L^*_{correlation} \leftarrow 0$

2.2.26.8.2. Traverse all individuals in the population $Populations_s$ using the index j. Note that if an individual record is skipped during the loop, the program returns to this point for the next individual:



2.2.26.8.2.1. Find the cell index $\mathbf{Index} = (Index_1, Index_2 ..., Index_{MaxDim})$ that corresponds to the data in the population set by traversing all Model table columns $d_i \in \mathbf{D}$ where $i = 1..MaxDim$ and performing the following:

2.2.26.8.2.1.1. If $d_i \notin Populations_s.Dimensions \cup 'Dummy'$ then:

2.2.26.8.2.1.1.1. Raise an error indicating the population set does not have sufficient information to support this operation.

2.2.26.8.2.1.2. If $d_i = 'Dummy'$ then:

2.2.26.8.2.1.2.1. Set the cell index component in the index vector to 1 since there is only one cell in the Dummy dimension, i.e. $Index_i = 1$

2.2.26.8.2.1.3. Else, meaning that $d_i \neq 'Dummy'$, then:

2.2.26.8.2.1.3.1. If information is missing for this dimension in this individual, i.e. $Populations_s(j, d_i) = \phi \wedge d_i \neq 'Dummy'$, then:

2.2.26.8.2.1.3.1.1. Skip the processing of this individual record by returning back to the loop for individual j and start processing on next individual. This is a return point similar to the 'continue' command in a 'for' loop then.

2.2.26.8.2.1.3.2. Init the range index $k = 1$

2.2.26.8.2.1.3.3. While $Populations_s(j, d_i) > d_i.Range_{k+1}$ meaning that the value is not in the cell:

2.2.26.8.2.1.3.3.1. Advance the range index for this dimension $k = k + 1$

2.2.26.8.2.1.3.3.2. If $k = |d_i.Range|$ then Indicate to the user with an error that the value in the population set is out of the table bounds.

2.2.26.8.2.1.3.4. After the index has been found check if the table range is categorical and the value was valid. i.e. If $d_i.Range_1 = NaN \wedge Populations_s(j, d_i) \neq d_i.Range_{k+1}$ then Raise an error indicating that the value in the population set does not fit the categorical nature of the data.

2.2.26.8.2.1.3.5. Set the cell index component in the index vector, i.e. $Index_i = k$

2.2.26.8.2.2. Knowing the Cell index Calculate the probability expression L^*_{model} by substituting the current cell values in

the symbolic expression string contained in the temporary model probability coefficient table

$L^*_{\mathrm{model}} = TempModelCoefTable(\mathbf{Index})$

2.2.26.8.2.3. Construct the study correlation term for the cell L^*_{study} by using the regression equation $\varphi(\boldsymbol{\gamma},\mathbf{a})$ that is written in terms of multiplication of the system reserved temporary coefficient vector $\boldsymbol{\gamma}$ and the parameter vector $\mathbf{a} = t_{jk}.RegressionParamVector$, which is generally composed of covariates and similar types, both of length n^*. It is defined in the study in the regression type:

Construct the **Regression Study Log Likelihood** term

2.2.26.8.2.3.1. If regression type is exponential then $\varphi(\boldsymbol{\gamma},\mathbf{a}) = -\boldsymbol{\gamma}\mathbf{a}$.

2.2.26.8.2.3.2. Else if regression type is linear then $\varphi(\boldsymbol{\gamma},\mathbf{a}) = \boldsymbol{\gamma}\mathbf{a}$.

2.2.26.8.2.3.3. Else if regression type is UKPDS then:

2.2.26.8.2.3.3.1. If $a_{n^*} \neq Studies_s.StudyLength$ then raise and error signifying that the study should have UKPDS form that requires time. Note that this member in the parameter vector should be a constant rather than a parameter.

2.2.26.8.2.3.3.2. Construct the formula $\varphi(\boldsymbol{\gamma},\mathbf{a}) = 1 - e^{-\left(\frac{1-\left(\gamma_{n^*}\right)^{a_{n^*}}}{1-\gamma_{n^*}}\right)\prod_{k=1}^{n^*-1}\gamma_k{}^{a_k}}$ Note that γ_{n^*} of the vector $\boldsymbol{\gamma}$ gets a special treatment.

2.2.26.8.2.3.4. Else if regression type is One minus exponential then $\varphi(\boldsymbol{\gamma},\mathbf{a}) = 1 - e^{-\boldsymbol{\gamma}\mathbf{a}}$.

2.2.26.8.2.4. For each member a_i of $\mathbf{a}$ calculate a value v_i by traversing the vector of parameters with index i and performing the following:

2.2.26.8.2.4.1.1. If a_i is a constant then $v_i = a_i$

2.2.26.8.2.4.1.2. Else if a_i is a parameter then:

2.2.26.8.2.4.1.2.1. If a_i is not represented in the population set then, i.e. if $a_i \notin Population_s$ then Raise an error indicating the population is not compatible with the model and is missing a parameter.

2.2.26.8.2.4.1.2.2. Find the value of the individual in this dimension $v_i = Populations_s(j, a_i)$

2.2.26.8.2.4.1.2.3. If no value was given, i.e. if $v_i = \phi$ then skip the processing of this individual record by returning back to the loop for individual j and start processing on next individual. This is a return point similar to the 'continue' command in a 'for' loop.

2.2.26.8.2.5. Exchange the vector **a** with the vector **v** composed of the values v_i to receive the correlation study term for this individual , i.e. $L^*_{study\_Unsubstituted}(\boldsymbol{\gamma}) = \varphi(\boldsymbol{\gamma}, \mathbf{v})$

2.2.26.8.2.6. Increase the count of valid individuals in the correct cell i.e.
$TempIndividualCount(\mathbf{Index}) = TempIndividualCount(\mathbf{Index}) + 1$

2.2.26.8.2.7. Accumulate the Study correlation expressions for permutations permutation of $\boldsymbol{\gamma}$ by traversing values $c = 0..2n^*$ and performing the following:

2.2.26.8.2.7.1. Generate the vector **u** of size n^* such that $u_c = h$, $u_{c-n^*} = -h$ and $u_i = 0 \forall i \neq c \wedge i \neq c - n^*$. Note that h has a small perturbation for calculating a numeric derivative. It is defined by the user as input, or defined as a constant and depends on machine precision. Also note that only one member in **u** may be non zero and if $c > n^*$, it will be negative.

2.2.26.8.2.7.2. Substitute the system coefficient vector members $\boldsymbol{\gamma}$ with their expected values $\hat{\boldsymbol{\lambda}} = t_{jk}.RegressionCoefVector$ after perturbation with **u** in the equation vector to receive the Study correlation term, i.e. $L^c_{study} = L^*_{study\_Unsubstituted}(\hat{\boldsymbol{\lambda}} + \mathbf{u})$. Note that this turns the term into a constant for this individual. Also note that it is possible to make this substitution before individual parameters have been substituted. However, in this case a more complicated data structure is required to hold the results for all perturbations. The current implementation is described here this way for simplicity.

2.2.26.8.2.7.3. Accumulate the study correlation term in the correct cell of the correct temporary study term accumulation table. i.e. $TempStudyTermAccTable_c(\mathbf{Index}) = TempStudyTermAccTable_c(\mathbf{Index}) + L^c_{study}$

Note also that $L^c_{correlation}$ means the correlation with a

small perturbation in location c and $L^0_{correlation}$ means no perturbation.

2.2.26.9. Else if the population set for the Study is defined categorically by distribution functions then:

2.2.26.9.1. Raise an Error as this is not supported.

Construct the translated **Regression Log Likelihood**

2.2.26.10. Determine the number of coefficients m^*. This number will represent the number of unknowns and will determine the size of matrices later on – this number is actually the number of cells in the table. From now on the different coefficients will be noted as $\mathbf{x} = \begin{bmatrix} x_1 & \cdots & x_{m^*} \end{bmatrix}$. Note that these coefficients are actually defined as $\eta_{Index_1\_Index_2\_\ldots\_Index_{MaxDim}}$ indicating a location in the model table.

2.2.26.11. Define the expected value coefficient vector $\bar{\mathbf{x}}$ of size m^*

2.2.26.12. Define the forward perturbation solution matrix $\bar{\mathbf{x}}^+$ of size $m^* x n^*$

2.2.26.13. Define the backward perturbation solution matrix $\bar{\mathbf{x}}^-$ of size $m^* x n^*$

2.2.26.14. Traverse all cells defined in $\mathbf{D}$ by recursively traversing all dimensions. For each table cell index $i..m^*$ that corresponds to the index set $\mathbf{Index} = (Index_1, Index_2 ..., Index_{MaxDim})$ perform the following:

2.2.26.14.1. If $TempIndividualCount(\mathbf{Index}) = 0$ raise an error indicating that there is not sufficient information in the population to calculate regression information for that cell in the table.

2.2.26.14.2. Otherwise for each perturbation $c = 0..2n^*$ perform the following:

2.2.26.14.2.1. Calculate the correlation solution for this perturbation, $\mu = \dfrac{TempStudyTermAccTable_c(\mathbf{Index})}{TempIndividualCount(\mathbf{Index})}$.

2.2.26.14.2.2. Store this value in the appropriate matrix/vector by checking c:

2.2.26.14.2.3. if $c = 0$, meaning no perturbation then:

2.2.26.14.2.3.1. Define the expected value solution as: $\bar{x}_i = \mu$. Note that this builds $\bar{\mathbf{x}}$.

2.2.26.14.2.4. else if $0 < c \le n^*$, meaning forward perturbation then:

2.2.26.14.2.4.1. Define the forward perturbation solution matrix $\bar{\mathbf{x}}^+$ member $\bar{x}^+_{i,c} = \mu$

2.2.26.14.2.4.2. Define the backwards perturbation solution matrix $\bar{\mathbf{x}}^-$ member $\bar{x}^-_{i,c-n^*} = \mu$

2.2.26.14.3. Construct the Jacobian $\mathbf{J}$ by performing the following $\mathbf{J} = \left(\bar{\mathbf{x}}^+ - \bar{\mathbf{x}}^-\right)/(2h)$. Note that central differences of order $O\left(h^2\right)$ are used and that the Jacobian is of size $m^* x n^*$.

2.2.26.15. Calculate the new covariance matrix $\mathbf{\Psi}$ associated with $\mathbf{x}$ by performing the following matrix multiplication $\mathbf{\Psi} = \mathbf{J} * \mathbf{\Sigma} * \mathbf{J}^T$. Note that $\mathbf{\Psi}$ is of size $m^* x m^*$.

2.2.26.16. Construct the un-substituted translated regression expression $L^*_u(\mathbf{x}, \bar{\mathbf{x}})$ using the model coefficient values vector $\mathbf{x}$ of length m^*, and the new covariance matrix $\mathbf{\Psi}$ of size $m^* x m^*$.

$$L^*_u(\mathbf{x}, \bar{\mathbf{x}}) = \log\left((2\pi)^{-m^*/2} \left|\mathbf{\Psi}\right|^{-1/2} e^{-\frac{1}{2}(\mathbf{x}-\bar{\mathbf{x}})\ \mathbf{\Psi}^{-1}(\mathbf{x}-\bar{\mathbf{x}})} \right) =$$

$$-m^*/2 * \log(2\pi) - 1/2 * \log(\left|\mathbf{\Psi}\right|) - \frac{1}{2}(\mathbf{x}-\bar{\mathbf{x}})\ \mathbf{\Psi}^{-1}(\mathbf{x}-\bar{\mathbf{x}})$$

2.2.26.17. Reset the actual model coefficient vector ζ to a vector of size m^*

2.2.26.18. If $\mathbf{D} = \{'Dummy'\}$ then:

2.2.26.18.1. There should be only one parameter in the vector $x = \eta$, in which case, $\zeta = \left[y^\tau_{jk}\right]$ where y^τ_{jk} is a member from probability matrix $\mathbf{Y}^\tau$ for time $\tau = Sudies_s.Time$ and for this transition. Note that an assertion check can be made here to verify if there is only one parameter used in the expression.

2.2.26.19. Else if $\mathbf{D} \neq \{'Dummy'\}$ then:

2.2.26.19.1. Create the actual model coefficient ζ in the likelihood by traversing the vector positions $i = 1..m^*$ and performing the following:

2.2.26.19.1.1. Since x_i represents a coefficient of the form $\eta_{Index_1\_Index_2\_...\_Index_{MaxDim}}$ then extract the table index $\mathbf{Index} = (Index_1, Index_2 ..., Index_{MaxDim})$ from the symbolic coefficient name. Note that an assertion check can be made on the coefficient and its index at this point to verify validity.

2.2.26.19.1.1.1. Calculate the model expression to be replaced y^* by substituting the current cell values in the symbolic expression string contained by y^τ_{jk} in the probability matrix $\mathbf{Y}^\tau$ for time $\tau = Sudies_s.Time$. This is performed by calling the following function $\{y^*, \mathbf{T}\} = \text{ParseMarkovTerm}(y^\tau_{jk}, \mathbf{Index})$

2.2.26.19.1.1.2. Update the actual model coefficient $\boldsymbol{\zeta}$ with the newly calculated model expression so that $\zeta_i = y^*$

2.2.26.20. Substitute the coefficients in the un-substituted translated likelihood expression with the new actual model coefficient. $L_t^* = L_u^*(\mathbf{x} = \boldsymbol{\zeta}, \bar{\mathbf{x}})$

2.2.26.21. Update the study Log Likelihood L^* by adding the substituted translated transition contribution, i.e. $L^* \leftarrow L^* + L_t^*$

2.2.27. Else if the study transitions are defined by categorical transitions, then construct the study Log Likelihood expression L^* which will be composed of categorical data in this case:

Calculate the **Log Likelihood** term associated with a **Categorical Study**

2.2.27.1. Build the Log Likelihood function by traversing all transitions in the study and for each transition $t_{jk} \in Studies_s$ perform the following:

2.2.27.1.1. If the transition has meaningful categorical information or observed counts. Meaning that it does not represent a retrospective study transition and it is not used to signify initial observations before transitions, i.e. $t_{jk}.Probability \neq \phi \wedge k \neq NULL$ then perform the following:

2.2.27.1.1.1. Construct L^* by traversing all cells defined in $\mathbf{D}_s$ by recursively traversing all dimensions. The transition table and the cell will be determined later, However since the table structure was previously calculated, it is possible to traverse the indices. For each table cell index set $\mathbf{Index}^* = (Index_1^*, Index_1^* ..., Index_{MaxStudyDim}^*)$, perform the following:

2.2.27.1.1.1.1. Set up the previous probability term and the previous observed data count to zero $y_{Old}^* = 0, x_{Old}^* = 0$

2.2.27.1.1.1.2. Loop through all outcome times that were defined during preprocessing the study transition in chronological order i.e. $\tau \in t_{jk}.TimeSteps$ and perform the following:

2.2.27.1.1.1.2.1. Reset the model expression for the study cell accumulator $y^* = 0$

2.2.27.1.1.1.2.2. Loop through all model indices associated with the study cell as defined in the index mapping table, i.e. loop through all indices $\mathbf{Index}$ such that $\mathbf{Index} \in t_{jk}.\mathbf{MapIndexTable}(\mathbf{Index}^*)$

2.2.27.1.1.1.2.2.1. Calculate the probability expression y^{**} by substituting the current cell values in the

symbolic expression string contained by y^{τ}_{jk} in the probability matrix $\mathbf{Y}^{\tau}$ for time τ. This is performed by calling the following function

$\{y^{**}, \mathbf{T}\} = \text{ParseMarkovTerm}(y^{\tau}_{jk}, \mathbf{Index})$

2.2.27.1.1.1.2.2.2. Extract the model prevalence from the prevalence table, i.e.

$w = t_{jk}.\mathbf{MapCellPrevalenceTable}(\mathbf{Index})$

2.2.27.1.1.1.2.2.3. Append the model expression to the accumulator, i.e. $y^{*} \leftarrow y^{*} + wy^{**}$

2.2.27.1.1.1.2.3. Extract the observed count for this time and cell from the study table

$x^{*} = t_{jk}.Data(Index^{*}_{1}, Index^{*}_{2}..., Index^{*}_{MaxStudyDim}, TimeIndex).Value$

Note that this table contains a time dimension in addition to all other dimensions in $\mathbf{D}_s$ and it is defined as the last dimension to avoid complications.

2.2.27.1.1.1.2.4. Update the study Log Likelihood by adding the following term:

$L^{*} \leftarrow L^{*} + \left(x^{*} - x^{*}_{Old}\right) * \log\left(y^{*} - y^{*}_{Old}\right)$

Note that if $\left(x^{*} - x^{*}_{Old}\right) = 0$ we ignore this update so as to skip cells with no observed changes where prevalence may be NaN.

2.2.27.1.1.1.2.5. Update the previous probability term and the previous observed data count to the current values

$y^{*}_{Old} = y^{*}, x^{*}_{Old} = x^{*}$

2.2.27.1.1.1.3. Extract the study sample size for this cell by accessing the table in the transition

$n^{*} = t_{j\phi}.Data(Index^{*}_{1}, Index^{*}_{2}..., Index^{*}_{MaxStudyDim}).Value$

where the starting state is j and the ending state is $\phi = NULL$. Note that this table does not contain the time dimension as it represents the starting population for each dimension rather than progression.

2.2.27.1.1.1.4. Update the study Log Likelihood with the following term: $L^{*} \leftarrow L^{*} + \left(n^{*} - x^{*}_{Old}\right) * \log\left(1 - y^{*}_{Old}\right)$

Note that if $\left(n^{*} - x^{*}_{Old}\right) = 0$ we ignore this update so as to skip cells with no observed changes where prevalence may be NaN.

End of **Log Likelihood** Construction and the **Study** loop

A consistency check for all **Studies**

2.2.28. Update the project Log Likelihood function by adding the study Log Likelihood expression: $L \leftarrow L + L^*$

3. For each study with index $s = 1..|Studies|$:
   3.1. Check if study $Studies_s$ was used and if not, raise a warning message to the user signifying that this study is incompatible with the model.
4. Optimize the function $L \rightarrow \min$ w.r.t. coefficient values.