PIANOS design document

Group Linja

Helsinki 7th September 2005

Software Engineering Project

UNIVERSITY OF HELSINKI
Department of Computer Science

Course

581260 Software Engineering Project (6 cr)

Project Group

Joonas Kukkonen
Marja Hassinen
Eemil Lagerspetz

Client

Marko Salmenkivi

Project Masters

Juha Taina
Vesa Vainio (Instructor)

Homepage

`http://www.cs.helsinki.fi/group/linja`

Contents

1 Overview

This is a design document of the PIANOS project.

This document describes the design of the software corresponding to the Software requirements specification document written earlier.

The software is implemented as a generator, which reads the input files (for example, the model description) and generates a Fortran program which solves the particular simulation problem associated with the particular model. The generator is implemented in Java.

A Fortran prototype which simulates an example model has also been implemented at the design phase.

1.1 Chapters

1. Overview	Describes this document's purpose.
2. Glossary	Explains the glossary used in this document.
3. Architecture diagram	Describes the division of the software into the generator and the executable program the generator produces.
4. The formats of input files	Describes the formats of the model description language and simulation parameter input files.
5. NAG library	Describes the useful parts of the NAG library and their correspondence to the distributions mentioned in the SRS.
6. Generated program	Defines the modules, subroutines and functions of the generated program which the generator produces.
7. Generator	Describes the generator at a general level.
8. Data structures	Describes the data structures used in the generator program.
9. Modules	Describes the modules of the generator program.
10. Algorithms	Describes the algorithms the generator uses.
11. Requirements	Describes which requirements are implemented and which chapters in this document correspond to them.

1.2 Version history

Version	Date	Modifications
0.2	15.06.2005	The document template
1.0	25.07.2005	First complete draft
1.1	03.08.2005	Reviewed and corrected final

2 Glossary

2.1 Probabilistic inference

Random variable (synonym: stochastic variable): Function which combines events with their probabilities. A numeric variable related to a random phenomenon (for example throwing a dice). The value of the variable is determined if the result of the phenomenon is known, otherwise only the probabilities of different values are known. [Tode04]

Discrete random variable: Random variable with a discrete range.

Point probability function: (Synonym: frequency function) Function $f : R \rightarrow R$ related to discrete random variable X so that $\forall x \in R : f(x) = P\{X = x\} =$ the probability that the value of X is x. [Tode04]

Density function: $f(x)$ is a density function iff

1. $f \geq 0$
2. f is integrable in R and $\int_{-\infty}^{\infty} f(x)dx = 1$.

[Tode04]

Cumulative distribution function: The function $F : R \rightarrow R$ is the cumulative distribution function associated with random variable X iff $F(x) = P\{X \leq x\} =$ the probability that X is less than or equal to x. [Tode04]

Continuous distribution: A random variable has a continuous distribution as its density function f if $\forall a, b \in R : P\{a \leq X \leq b\} = \int_a^b f(x)dx$ [Tode04]

Joint distribution: If X and Y are random variables, their joint distribution describes how probable all the possible combinations of X and Y are.

Independence: Events A and B are independent, if the probability of B is independent on whether A has happened or not. (For example if A = "it rains", B = "when I throw a dice the result is 6" and C="when I throw a dice the result is even" then A and B are independent but B and C are not.) [Tode04]

Conditional probability: If X and Y are random variables, the conditional probability $P\{X = x|Y = y\}$ means the probability that the value of X is x if it is assumed that the value of Y is y.

Bayes's rule: $P\{X = x|Y = y\} = \frac{P\{Y=y|X=x\} \cdot P\{X=x\}}{P\{Y=y\}}$. The rule is obtained from the observation that $P\{Y = y|X = x\} \cdot P\{X = x\} = P\{X = x \text{ AND } Y = y\}$

Chaining: Using the Bayes's rule many times consecutively.

Bayesian model: A model which connects dependent random variables to each other by defining dependencies and conditional probabilities. The model defines the joint distribution of the variables.

Likelihood function: $P\{data|explanation\}$ is called the likelihood function because it defines how likely it is to get such a data if the real conditions are known. (For example if

we know that a bird resides at some area, then how likely it is to get the observations we have already got.)

Markov random field: A random field that exhibits the Markovian property: $\pi(X_i = x_i|X_j = x_j, i \neq j) = \pi(X_i = x_i|\delta_i)$, Where δ_i is the set of neighbours for X_i. That is, only the adjacent units affect the conditional probability. [Rand]

Discrete distributions:

- **Uniform distribution**
- **Binomial distribution**
- **Geometric distribution**
- **Poisson distribution**

Continuous distributions:

- **Uniform distribution**
- **Normal distribution**
- **Multinomial distribution**
- **Exponential distribution**
- **Binormal distribution**
- **Lognormal distribution**
- **Beta distribution**
- **Gamma distribution**
- **Dirichlet distribution**

Descriptions of these distributions can be found in [Math05].

Functional dependence: A condition between two variables X and Y so that the value of X determines the value of Y unambiguously.

Stochastic dependence: A condition between two variables X and Y so that the value of X doesn't determine the value of Y but influences the probabilities of the possible values.

Spatial dependence: A special case of stochastic dependence where the dependence is related to some spatial structure. (For example towns that are adjacent to each other influence to each other.) The spatially dependent variables form a Markov random field.

Prior distribution: The prior distribution of the parameters describes their assumed joint probability distribution before inferences based on the data are made.

Posterior distribution: The posterior distribution of the parameters describes their joint probability distribution after inferences based on the data are made.

Marginal distribution: The prior or posterior distribution concerning only one parameter.

2.2 Models and related concepts

Model: Means: Bayesian model

Variable: A variable is an entity in the model that can have an assigned value from its range of values. Variables and their dependencies form the base of the problem that the software is developed to solve.

Parameter: A parameter is a variable whose value is not defined by data.

Adjacency matrix: The adjacency matrix of a simple graph is a matrix with rows and columns labeled by graph vertices, with a 1 or 0 in position ij according to whether i and j are adjacent or not.

Floating point number: A computer representation of a real number with finite precision.

2.3 Metropolis-Hastings algorithm

Iteration: A single round of the algorithm when all the parameters have been updated once.

Burn-in-iterations: The iterations that are run before any output is produced.

Thinning factor: The thinning factor t means that every t^{th} iteration value is used in the output and the rest are discarded.

Block: A set of parameters which are defined to be updated together. That is, the proposals are generated to all of them and the acceptance of all the proposals is decided at the same time.

Update: Proposing a value to a parameter and then accepting it (the value changes) or discarding it (the value remains the same).

Proposal strategy: The proposal strategy defines how the next proposed value is generated. Possible choises are

1. Fixed proposal strategy: The next proposed values for a parameter is taken from its proposal distribution.

2. Random walk: The next proposed value for a parameter is created by adding a value taken from the proposal distribution to the current value of the parameter.

Proposal distribution:

1. The distribution from which the next proposed value for a parameter is chosen (when using the "Fixed proposal distribution" proposal strategy).

2. The distribution that is used in generating proposed values for a parameter by adding a value taken from the distribution to the parameter's current value (when using the "Random walk" proposal strategy).

Update strategy: The update strategy describes which variables belong to the same block. (See: Block) The update strategy also includes information about whether the blocks are considered for updates in sequential order, whether the next block to update is chosen at random or whether the block to update is chosen based on the block weights.

Convergence: The phenomenon that during the simulation the parameter values get closer to the posterior distribution. The speed of the convergence depends on the initial values and other simulation parameters.

2.4 Naming conventions

Fortran: Refers to the fortran programming language, version 90/95 specifically. **fortran** refers to the whole Fortran family, and **FORTRAN** refers to FORTRAN/77 specifically.

Proposal: A new value cancidate obtained from the proposal distribution.

Frequency function is used to refer to a frequency function or a density function when it is irrelevant which one there really is, that is, when it's irrelevant whether the distribution is discrete or continuous.

Variable is used to refer a variable or a parameter when it's irrelevant which one there really is.

Generator: Used to refer to the modules of the software that write out the specific executable Program that carries out the simulation for a given simulation model.

Program: The program that is run to simulate the problem model. Synonym: generated program.

Entity: A data structure of the Generator representing a repetitive structure (indexing structure) of variables. For example $alpha_i$, x_i both are part of the Entity indexed with i.

Variable group: All variables of a group, that is $alpha_i$ for all i.

Parser: The ComputationalModelParser class used for reading the input files for the Generator.

3 Architecture diagram

This section contains a diagram of the division of the software into the generator and the executable program the generator produces.

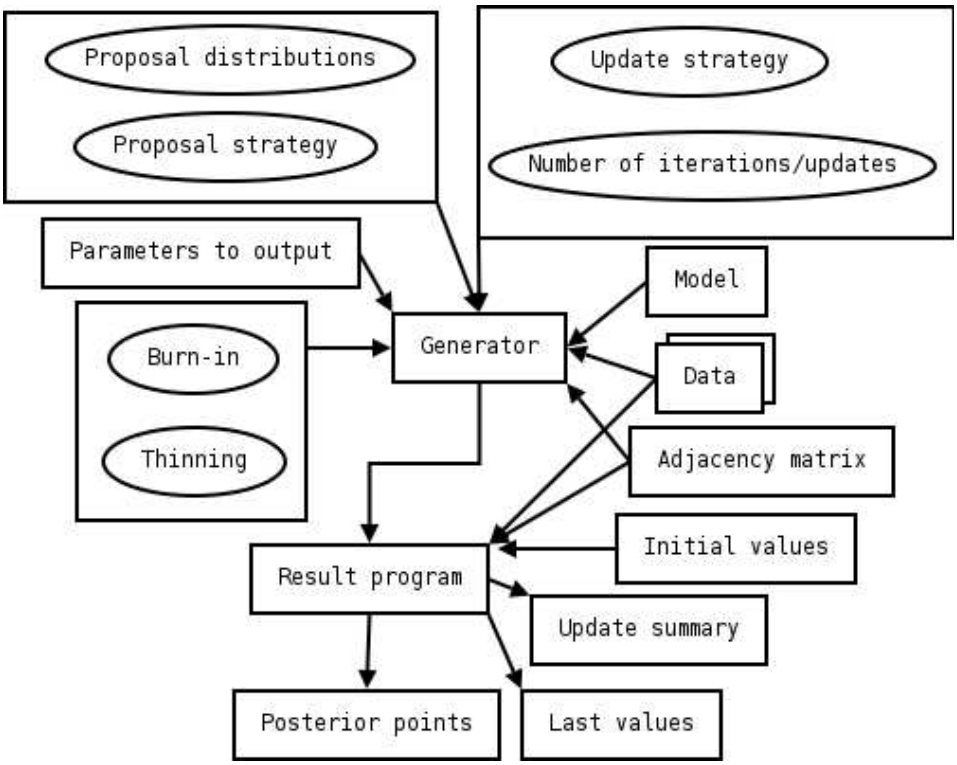

Figure 1: A diagram of the software input/output files and data flow.

Figure 1 represents the generator and the result program that it produces, and how the data files are used by them. Every rectangle is a file (except for the generator and the result program). An arrow means that data from the component at the source of the arrow is used by the component that the arrow leads to.

4 The format of input files

This section describes the format of the input files, that is the model description and the files for the simulation technical parameters.

The input files are:

- The model (including names of data files)
- The burn-in iteration length and the thinning factor
- Proposal distributions adn proposal strategies
- Initial values
- Update strategy and the number of updates
- Parameters to output

4.1 Model description language

A model description consists of variable and entity definitions. It can also include comment lines, any line beginning with '#' is considered a comment.

The model description file consists of the following parts:

1. Definitions of global variables
2. Definitions of entities, which contain definitions of variables

4.1.1 Names and constants

Names of variables and entities can include lowercase letters 'a' - 'z'. No other characters are allowed. The variable names must not be equal reserved words of Fortran.

Floating point constants must be expressed by using the scientific syntax except that the exponent "E" must be an uppercase "E".

4.1.2 Defining variables

Variables can be either integers or floating point numbers. This is expressed by an 'INTEGER' or 'REAL' in front of their names.

If the variable is functional, an equation must be provided. If the variable is stochastic, a distribution must be provided. The syntax is:

```
type name = expression
type name ~ distribution
```

A variable must be defined before it can be used in other variables' equations or distributions.

List of distributions

The following distributions are possible for INTEGER variables:

- discrete_uniform(INTEGER a, INTEGER b)
- binomial(INTEGER n, REAL p)
- poisson(REAL mu)

The following distributions are possible for REAL variables:

- continuous_uniform(REAL a, REAL b)
- beta(REAL alpha, REAL beta)

Examples:

```
REAL alpha ~ beta(0.1, 1.0)
INTEGER gamma ~ poisson(alpha)
```

List of operators allowed in equations

1) Functional parameters describing spatial relations

A special character & has a meaning "all neighbour units". & x means "all x instances in neighbour units".

The following operations are possible:

- SUM(&variable)
- COUNT(&variable)

Note that functional parameters describing spatial relations can only appear inside entities. Note also that the expressions SUM and COUNT cannot appear inside other expressions.

Examples:

```
REAL q=SUM(&x)
INTEGER c=COUNT(&x)
REAL p = q / c
# the average of x in the neighbour units
```

2) Other functional parameters

The following operations are possible:

- +

- -
- *
- /
- ** (power expression in Fortran)
- EXP()
- LOG()

Examples:

```
REAL beta = alpha**2
REAL gamma = EXP(alpha + 1) / EXP(alpha - 1)
```

4.1.3 Defining entities

The first line, identified with ENTITY, defines the name of the entity, the data file where all data about the entity is found (if some variables from the entity are taken from data), the data name defining spatial relations (if exists) and which other entities the entity combines (if the entity describes an intersection of two entities).

The variables related to the entity are defined inside the brackets. The variables are defined as earlier except that is is possible to define where the variable is found in the data. This is done by adding "(column_number)" after the variable name, where column number defines the column of the data file in which the corresponding data is found. If the data is a matrix, there is no single column containing the data and this situation is specified simply by adding "(*)" after the variable name.

Examples:

```
ENTITY bird, "birds.txt"
{
    REAL size(1)
}
```

In this example there is an entity called bird. It contains one variable which is taken from data. The data is found in file birds.txt at column 1.

If no data is provided, the data file name must equal "". It is also possible that an entity has no variables in it. For example:

```
ENTITY bird, ""
{

}
\end{verbatim
```

```
ENTITY square, "squares.txt", "spatial.txt"
{
    INTEGER northerness(1)
}
```

Similarly we define an entity called square. In this example square is a spatial unit, so a file name of the adjacency matrix is given.

```
ENTITY observation, "observations.txt", combines(square, bird)
{
    REAL p = EXP(alpha * northerness) / (1 + EXP(alpha * northerness))
    INTEGER x ~ user_bernoulli(p)
    INTEGER obs(*) ~ user_defined_points(x)
}
```

This example shows how to define an intersection entity. The combines-clause states that the squares are the vertical units and birds are the horizontal units in the matrices related to the intersection entity (for example, data matrices).

Note that there can be only one intersection entity and it cannot be spatial.

4.1.4 Distribution defined by the user

User-defined distributions can be used like the other distributions. The names of user-defined distributions must begin with "user_".

For example:

```
x ~ user_bernoulli(p)
```

See also: 4.3

4.2 Simulation parameters as input

4.2.1 The burn-in length and the thinning factor

The burn-in length and the thinning factor are both integers. They must be given in a single file, given like in the example below:

```
?? simulation
? burn-in
1000

? thinning
4
```

The first line is a compulsory legend line.

4.2.2 Initial values

The user must provide initial values for all parameters. The initial values are given in a single file.

The first line of the file must be:

```
?? initial values
```

The format of the file is:

1) Legend, which can be one of the following:

```
? parametername

? parametername begin:end

? parametername begin:end begin2:end2
```

2) The initial values in an array or a matrix corresponding to the legend.

The file may contain several consecutive legend - values - pairs.

An example:

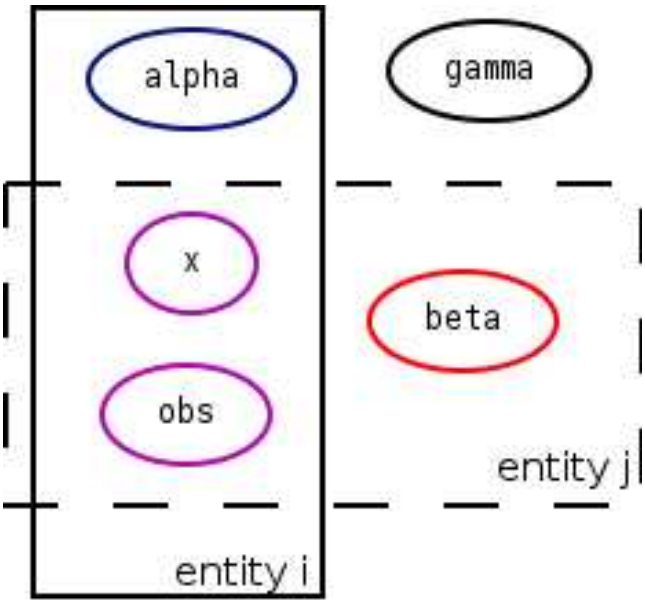

Figure 2: A model with two intersecting entities

```
? gamma
4.0
```

The legend means that we are giving an initial value to a single variable on the line after the legend.

```
? alpha 1:5
3.2
0.3
3.1
5.3
2.9
```

The legend describes that we are giving initial values to $alpha_1, alpha_2, alpha_3, alpha_4$ and $alpha_5$ in an array on the lines following the legend line. After setting the initial values, we have $alpha_1 = 3.2, alpha_2 = 0.3$ etc.

```
? beta 1:3
5.4
2.1
1.0
```

The legend means that we are giving initial values to $beta_1, beta_2$ and $beta_3$ in an array on the lines after the legend. After setting the initial values, we have $beta_1 = 5.4, beta_2 = 2.1$ etc.

```
? x 1:5 1:3
1.0 3.2 5.3
1.2 5.2 3.1
3.8 4.1 8.0
4.4 2.7 3.2
5.6 0.4 1.2
```

The legend describes that we are giving initial values to $x_{1,1}, x_{1,2}, x_{1,3}, x_{2,1}, ...x_{5,3}$. The values are in a matrix form after the legend line. The first index corresponds to the vertical dimension and the second index corresponds to the horizontal dimension of the matrix. After setting the initial values, we have $x_{3,2} = 4.1$ for example.

If some instances of, say, the variable obs are missing from the data, the user must provide initial values for them.

```
? obs 4:4 1:1
4.3
```

This definition states that the initial value of $obs_{4,1}$ is 4.3.

4.2.3 Proposal strategies and proposal distributions

The format of the proposal strategies and distributions input file is similar to the initial values input file.

The first line of the file must be:

```
?? proposal distributions
```

An example of giving the proposal distributions:

```
? alpha all
poisson(3.0)
```

This expression states that the parameters $alpha_1, ..., alpha_5$ have the same proposal distribution Poisson(3.0) and the fixed proposal strategy is used as default.

If the parameter is global, the word "all" is not needed. For example:

```
? gamma
continuous_uniform(0.0, 2.0)

? x all
continuous_uniform(-1.0, 1.0) RW
```

This expression states that the parameters $x_{1,1}, ...$ use the random walk as proposal strategy and the proposal distribution for them is continuous_uniform(-1.0, 1.0).

Possible distributions for INTEGER variables include:

- discrete_uniform(INTEGER a, INTEGER b)
- binomial(INTEGER n, REAL p)
- poisson(REAL mu)

Possible distributions for REAL variables include:

- continuous_uniform(REAL a, REAL b)
- beta(REAL alpha, REAL beta)

It is also possible to use a distribution defined by the user.

4.2.4 The update strategy and the number of updates

The first line of the specification file must be:

```
?? update
```

The format of the rest of the file depends on the update strategy to be chosen.

1) If the user wants to update the parameters in sequential order:

```
? strategy
sequential

? iterations
42
```

The user gives the number of iterations, that is: the parameters are updated once and then the output is printed (considering the thinning factor of course) and this is repeated until the iteration count is reached.

2) If the user wants the next parameter to update to be chosen at random:

```
? strategy
random

? updates
500

? x all
600
```

The user gives first gives the update strategy, then the default number of updates. Considering this example all parameters must be updated at least 500 times before the simulation is finished. After giving the default number, the user can also give the number of updates for a parameter group.

Note: It is possible to define the number of updates for a parameter group, not for a single parameter instance (for example, for x but not for $x_{1,6}$).

Note: The output is printed after each update so the user may want to use a bigger thinning factor.

4.2.5 Which parameters to output

The parameters to output are defined in a manner similar to the initial values. For example:

The compulsory legend is:

```
?? output outputFile summaryFile
```

where outputFile is the filename of the output file and summaryFile is the filename of the summary file.

```
? alpha all
? beta all
? gamma
```

This definition states that the parameter to output are all $alphas$, all $betas$ and $gamma$. Note that it is not possible to define that only some instances, for example $alpha_1$, are output.

4.3 Adding distributions

When the user wants to add an own distribution, the following steps are needed:

1. Decide a name for the distribution. The name must begin with "user_".
2. Add an explanation line in the beginning of user_dist.f90.
3. Add subroutine user_name_freq() into the module user_dist.f90. The parameters depend on the parameters of the distribution.
4. If the distribution could be used as a proposal distribution, add user_name_gen() into the module user_dist.f90.

The parameters of user_name_freq and user_name_gen subroutines depend on the parameters of the distribution: the subroutine implementing the frequency function has two additional parameters (the point on which the frequency is calculated and the result) and the subroutine generating the proposals has one additional parameter (the array which to generate proposals to).

Note that the generation subroutine is assumed to generate an array full of proposals.

Note also that the user might want to use the NAG library for existing generation subroutines, see chapter 5.

The explanation has the following syntax:

```
! name param_count types gen_present
```

Where

- "param_count" is an an integer
- "types" contain as many types as param_count suggests
- each type is either INTEGER or REAL

- gen_present is .TRUE. if there is a subroutine that generates proposals from the distribution, .FALSE. otherwise.

For example:

```
! user_defined_points 1 INTEGER .FALSE.
```

If the user wants to add many distributions all the explanations are written on consecutive lines at the beginning of the file. Note that no other comment lines are allowed at the beginning of the file.

5 NAG library functions

The correspondence between the NAG library subroutines / functions and the distributions included in the Requirements specification document [requirement M10].

5.1 Discrete distributions

Distribution	Frequency	Generation
Uniform distribution	(easy to calculate)	G05DYF
Binomial distribution	G01BJF	G05EDF and G05EYF
Geometric distribution		
Poisson distribution	G01BKF	G05DRF / G05ECF and G05EYF

5.2 Continuous distributions

Distribution	Density	Generation
Uniform distribution	(easy to calculate)	G05FAF
Normal distribution		G05FDF
Multinomial distribution		
Exponential distribution		G05FBF
Binormal distribution		
Lognormal distribution		G05DEF
Gamma distribution		G05FFF
Beta distribution	G01EEF	G05FEF
Dirichlet distribution		

The distributions missing either generation or density/frequency subroutines will be left to the user, as it has been agreed with the customer (unlike specified in the SRS).

6 Generated program

This section introduces the Program. It explains about the data structures of the Program, outlines its structure in modules and summarizes the operations of each module. The section provides deeper understanding of the simulation implementation, and serves as a reference for building the Generator.

6.1 Data structures

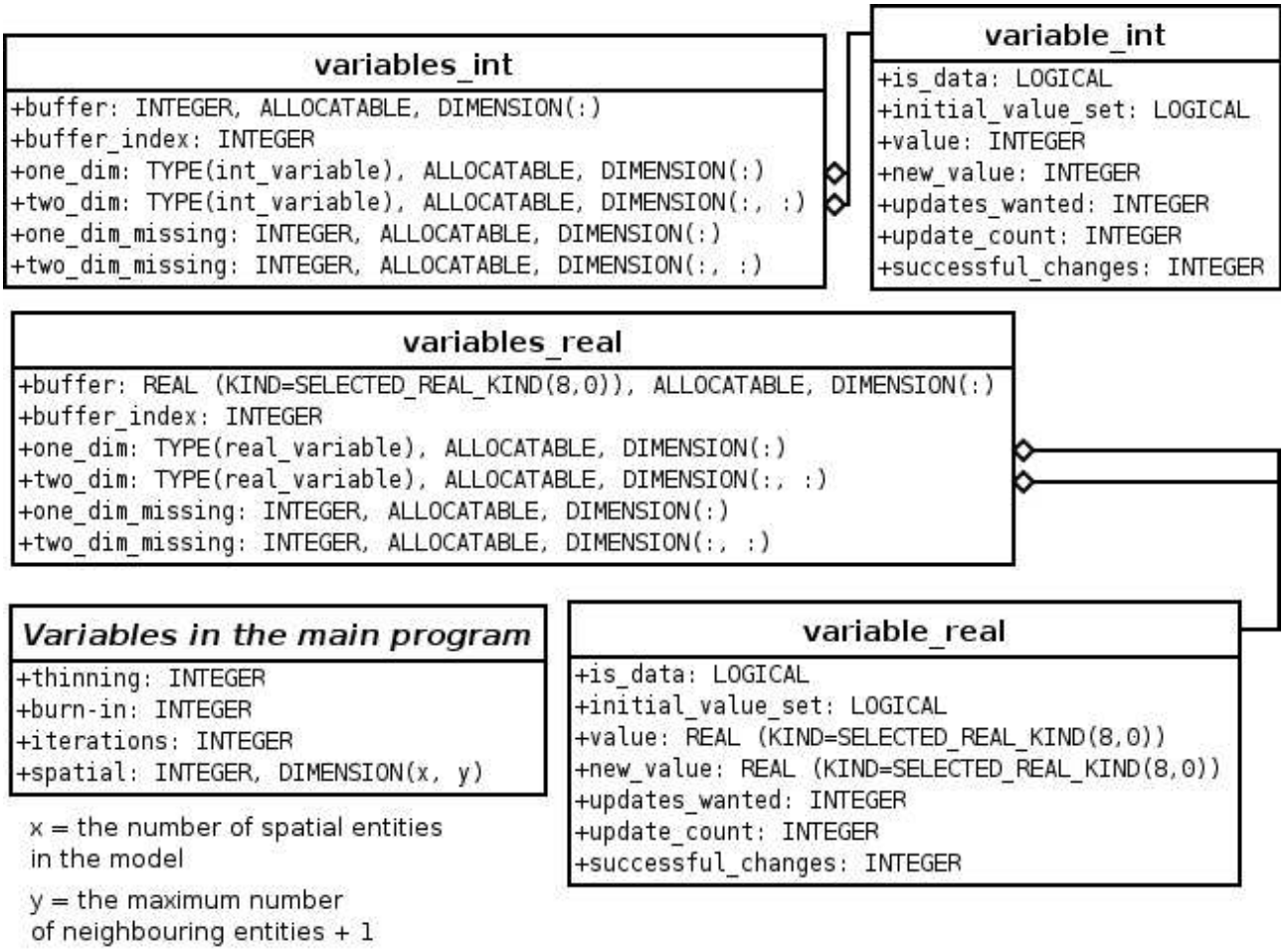

Figure 3: A diagram of the generated data structures.

Figure 3 shows the data structures used in the generated program. The *variable_int* and *variable_real* can represent both data variables and parameters. Each instance corresponds for example to one $alpha_{32}$ or $x_{31,4}$. A *variables* instance represents a repetitive structure of the model, for example $alpha$ and x. The *one_dim* and *two_dim* are used for one-dimensional and two-dimensional variable structures, respectively.

In the case that a variable comes from data that has missing values in it, the *one_dim_missing* or *two_dim_missing*-index array contains the indices of the missing data, which are then

treated as parameters.

The fields *update_count* and *updates_wanted* are included only if the user has chosen the random update strategy.

If the model has a spatial structure, it is represented with the help of the *spatial* array; this array has as many rows as the spatial structure has elements, and as many columns as is the greatest number of neighbours in the structure plus one. One row represents one spatial element. The first column of each row contains the number of neighbours that particular element has; the rest contain the indices of the neighbours.

6.2 Modules

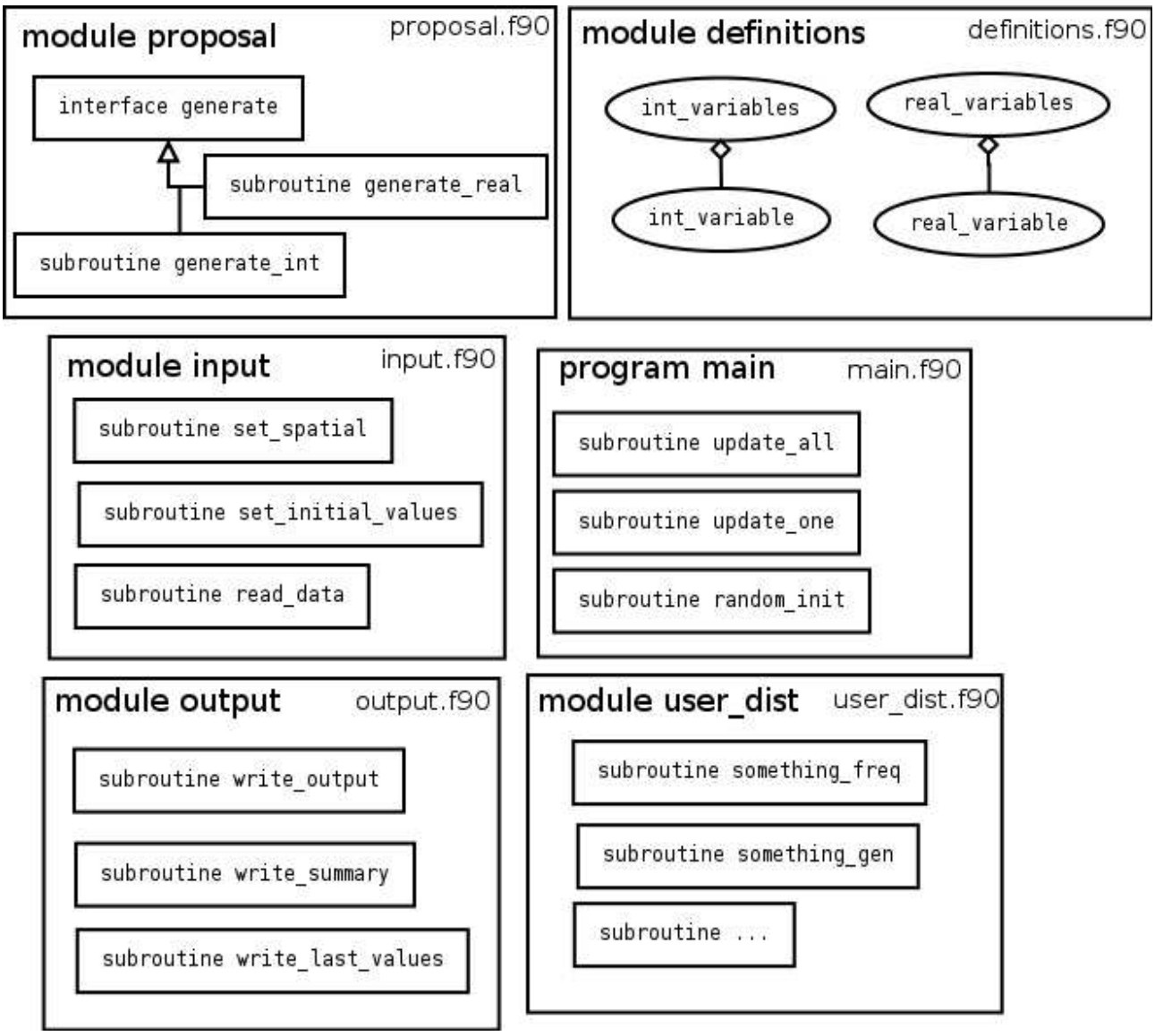

Figure 4: A diagram of the generated modules and their subroutines.

Figure 4 illustrates the division of the generated program into modules. Each module is placed in its own source file, which are indicated in the upper right corners.

For each subroutine and function the following information is included:

- *Subroutine/function name*: The name of the subroutine or the function in the program.
- *Description*: What the subroutine or the function is meant to do.
- *Parameters*: The names, types and intents (IN, OUT, INOUT) of the parameters.

- *Generating*: What information is needed when the subroutine or the function is generated.

This information should suffice for the implementation of the Generator and allow for quicker understanding of the prototype code and the final generated program structure.

6.2.1 Module proposal

Subroutine name:	generate_int
Description:	Generates a buffer of new proposals for a given variable by its name.
Parameters:	CHARACTER(LEN=*), INTENT(IN) :: name INTEGER, DIMENSION(:), INTENT(OUT) :: buffer 1. name: *On entry:* The name of the variable to generate proposals for, e. g. $alpha$ 2. buffer: *On exit:* The buffer filled with new proposals from the variable's proposal distribution.
Generating:	All the names of the variable groups, their proposal distributions (names and parameters) and the corresponding functions/subroutines.

Subroutine name:	generate_real
Description:	Generates a buffer of new proposals for a given variable by its name.
Parameters:	CHARACTER(LEN=*), INTENT(IN) :: name REAL, DIMENSION(:), INTENT(OUT) :: buffer 1. name: *On entry:* The name of the variable to generate proposals for, e. g. $alpha$ 2. buffer: *On exit:* The buffer filled with new proposals from the variable's proposal distribution.
Generating:	All the names of the variable groups, their proposal distributions (names and parameters) and the corresponding functions/subroutines.

6.2.2 Module input

Subroutine name:	read_data
Description:	Reads the data from data files into the data structure defined in figure 3.
Parameters:	The variables found in the model, for example: TYPE(variables_real), INTENT(INOUT) :: alpha TYPE(variables_int), INTENT(INOUT) :: beta TYPE(variables_int), INTENT(INOUT) :: x TYPE(variables_int), INTENT(INOUT) :: obs Note that the parameters depend on the model in use. 1. Each TYPE(variables_...): *On entry:* The data structure describing the corresponding variable in the model *On exit:* The same except the *is_data* and *value* fields are set.
Generating:	• The names of the data files (for each file the generator must generate a binary matrix describing which variables are not present in the data and generate a new data file where the 'no data' characters are replaced with zeros) • The names of the variables and whether they are one-dimensional or two-dimensional • The loop lengths, that is, how many different entities (birds, squares etc.) we have

Subroutine name:	set_initial_values
Description:	This subroutine reads the initial values from a data file into the data structure defined in figure 3.
Parameters:	The variables found in the model, for example: TYPE(variables_real), INTENT(INOUT) :: alpha TYPE(variables_int), INTENT(INOUT) :: beta TYPE(variables_int), INTENT(INOUT) :: x TYPE(variables_int), INTENT(INOUT) :: obs Note that the parameters depend on the model in use. 1. Each TYPE(variables_...): *On entry:* The data structure describing the corresponding variable in the model *On exit:* The same except the *value* fields are set corresponding the initial values.
Generating:	The name of the initial values file. The names of the variable groups and whether they are one-dimensional or two-dimensional.

Subroutine name:	set_spatial
Description:	This subroutine reads the adjacency matrix from a file and initializes the corresponding data structure.
Parameters:	The structure defining spatial relationships, for example: INTEGER, DIMENSION(300, 5), INTENT(IN) :: spatial Note that the parameters depend on the model in use. 1. INTEGER, DIMENSION(...) :: spatial: *On entry:* The data structure describing the spatial relationships *On exit:* The data structure correctly initialized. That is, spatial(i, 1) defines how many neighbours unit i (for example square i) has and spatial(i, 2) ... define the indices of the neighbours.
Generating:	The name of the adjacency matrix file. The type of the structure (the number of spatial units and the maximum number of neighbours).

6.2.3 Module output

Subroutine name:	write_output
Description:	This subroutine writes the output of one iteration into the output file. The file is opened and closed in the main program.
Parameters:	The variables found in the model, for example: TYPE(variables_real), INTENT(IN) :: alpha TYPE(variables_int), INTENT(IN) :: beta TYPE(variables_int), INTENT(IN) :: x TYPE(variables_int), INTENT(IN) :: obs Note that the parameters depend on the model in use. 1. Each TYPE(variables_...): *On entry:* The data structure describing the corresponding variable in the model
Generating:	Which parameters to write as output and whether they are one-dimensional or two-dimensional.

Subroutine name:	write_summary
Description:	This subroutine writes the summary of the simulation into a summary output file. The summary includes the number of updates and successful changes for each parameter.
Parameters:	The variables found in the model, for example: TYPE(variables_real), INTENT(IN) :: alpha TYPE(variables_int), INTENT(IN) :: beta TYPE(variables_int), INTENT(IN) :: x TYPE(variables_int), INTENT(IN) :: obs Note that the parameters depend on the model in use. 1. Each TYPE(variables_...): *On entry:* The data structure describing the corresponding variable in the model
Generating:	The names of the parameters and whether they are one-dimensional or two-dimensional. The file name of the summary file.

6.2.4 Program main

Program name:	main
Description:	This is the main program which performs the simulation by using the subroutines described below.
Generating:	• The names and types (real or integer) of the variables in the model • The name of the output file • The loop lengths, that is, how many different entities (birds, squares etc.) we have • The thinning factor and the burn-in iteration count • The update strategy • If the update strategy is 'sequential': the iteration count • If the update strategy is 'random': the count of wanted updates for each parameter • If the model is spatial: the maximum number of neighbours

Subroutine name:	random_init
Description:	This subroutine initializes the NAG random number generator.
Generating:	This subroutine is completely static so no information is needed.

Subroutine name: update_all

Description: This subroutine updates each parameter once. It occurs in the generated program if and only if the user has chosen the sequential update strategy.

Parameters: The variables found in the model and the structure defining spatial relationships, for example:
TYPE(variables_real), INTENT(INOUT) :: alpha
TYPE(variables_int), INTENT(INOUT) :: beta
TYPE(variables_int), INTENT(INOUT) :: x
TYPE(variables_int), INTENT(INOUT) :: obs
INTEGER, DIMENSION(300, 5), INTENT(IN) :: spatial
Note that the parameters depend on the model in use.

1. Each TYPE(variables_...):
   On entry: The data structure describing the corresponding variable in the model
   On exit: The same except the *value* fields are updated as the next iteration is performed.
2. INTEGER, DIMENSION(...) :: spatial:
   On entry: The data structure describing the spatial relationships

Generating:

- The names of the parameters and whether they are one-dimensional or two-dimensional
- The loop lengths, that is, how many different entities (birds, squares etc.) we have
- The proposal strategy (fixed or random walk) for each parameter
- The formula of the acceptance probability for each parameter (which variables depend on it and which parameter depend on it and what distributions define the dependencies)
- The proposal distributions for each parameter
- If the model is spatial: the maximum number of neighbours

Subroutine name: update_one

Description: This subroutine updates one parameter decided at random. It occurs in the generated program if and only if the user has chosen the random update strategy.

Parameters: The variables found in the model and the structure defining spatial relationships, for example:
TYPE(variables_real), INTENT(INOUT) :: alpha
TYPE(variables_int), INTENT(INOUT) :: beta
TYPE(variables_int), INTENT(INOUT) :: x
TYPE(variables_int), INTENT(INOUT) :: obs
INTEGER, DIMENSION(300, 5), INTENT(IN) :: spatial
Note that the parameters depend on the model in use.
INTEGER :: INTENT(INOUT) :: parameters_achieved

1. Each TYPE(variables_...):
   On entry: The data structure describing the corresponding variable in the model
   On exit: The same except the *value* fields are updated as the next iteration is performed
2. INTEGER, DIMENSION(...) :: spatial:
   On entry: The data structure describing the spatial relationships
3. parameters_achieved:
   On entry: The number of the parameters which have achieved their updates_wanted count
   On exit: The number of the parameters which have achieved their updates_wanted count

Generating:

- The names of the parameters and whether they are one-dimensional or two-dimensional
- The loop lengths, that is, how many different entities (birds, squares etc.) we have
- The proposal strategy (fixed or random walk) for each parameter
- The formula of the acceptance probability for each parameter (which variables depend on it and which parameter depend on it and what distributions define the dependencies)
- The proposal distributions for each parameter
- If the model is spatial: the maximum number of neighbours

7 Generator

The functionality of generator can be divided into following parts:

1. Create the DistributionFactory
2. Use the ComputationalModelParser to read input files
3. Link the objects
4. Generate the data structures and variable definitions
5. Generate the setting of initial values, reading of data and spatial matrix
6. Generate the printing of variables
7. Generate the proposal distribution module
8. Generate the main module
9. Generate the acceptance formulas for variables

With "generate" we mean generating and writing Fortran code using FortranWriter class.

Each part in more detail.

1. Create the DistributionFactory

A DistributionFactory object is created. The parser needs the object to determine which proposal distribution to set to a variable. DistributionFactory's constructor reads the user defined distributions file and constructs an object for each user-defined distribution.

2. Use the ComputationalModelParser to read input files

The model file, which defined variables and structures, is read first one line at a time. If the line describes a structure, an Entity object will be created. A structure definition ends with '}'. Name, file (if any), spatial matrix and some other data are set when an Entity object is constructed. The object will then be added to a collection. A variable can be global, or it can belong to an entity. Variables are created the same way as are the entities. The Parser processes the formula of each variable, creating either a Distribution or an Equation, accordingly.

Simulation parameters are read next. Global variables thinning, burn-in, etc are set. Proposal distributions are set to each variable. The parses uses DistributionFactory to determine the right distribution. A new Distribution is then created and added to the variable in question.

Lastly the data files are processed. All missing values are replaced with '0' and the program checks that each missing value has a value set in the initial values file. The Parser also writes a missing value matrix for each data file. The lengths of data files are saved in the related Entity objects. With the information gathered, the Parser creates a ComputationalModel instance and passes it to the Generator.

3. Link the objects

See 10.1.

4. Generate the data structures and variable definitions

The generateDefinitions() method is called. The Entity and Variable objects are examined and the data structures needed are coded into a Fortran module.

5. Generate the setting of initial values, reading of data and the spatial matrix

The generateInput() method is called. This method calls generateSetInitialValues(), generateReadData(), generateSetSpatial() which read the initial values, data and spatial matrices.

6. Generate the printing of variables

The generateOutput() method is called. This method calls generateWriteSummary(), and generateWriteOutput() methods. These methods generate the needed Fortran code for printing of variables.

7. Generate the proposal distribution module

The generateProposal() method is called. The method generates the proposal distribution module.

8. Generate the main module

The generateMain() method is called. The method generates the subroutines in the main program, the initialization of data structures and calls to other modules.

9. Generate the acceptance formulas for variables

This phase is done in the main module generation. The generateMain() method calls either generateUpdateAll() or generateUpdateOne() depending on the update strategy used. The methods code the update of variables. Both methods call the generateAcceptanceFormula() method, which generates the acceptance formula for a variable.

7.1 Operations of Generator

Operation	Return type	Description
generate()	void	The method doesn't generate any code itself. It calls other generate methods.
generateDefinitions()	void	This method generates the Fortran module Definitions and writes it into a source code file.
generateInput()	void	This method generates the Fortran module Input and writes it into a source code file.
generateOutput()	void	This method generates the Fortran module Output and writes it into a source code file. It calls the generateWriteSummary() and generateWriteOutput() methods.
generateProposal()	void	This method generates the Fortran module Proposal and writes it into a source code file.
generateMain()	void	This method generates the Fortran module Main and writes it into a source code file.
generateSetInitialValues()	String[]	The method generates the subroutine set_initial_values.
generateReadData()	String[]	The method generates the subroutine read_data.
generateSetSpatial()	String[]	The method generates the subroutine set_spatial.
generateWriteSummary()	String[]	The method generates the subroutine write_summary.
generateWriteOutput()	String[]	The method generates the subroutine write_output.
generateUpdateAll()	String[]	The method generates the subroutine update_all.
generateUpdateOne()	String[]	The method generates the subroutine update_one.
generateAcceptanceFormula (Variable variable)	String[]	The method generates and returns the acceptance formula for a given Variable object.

8 Data structures

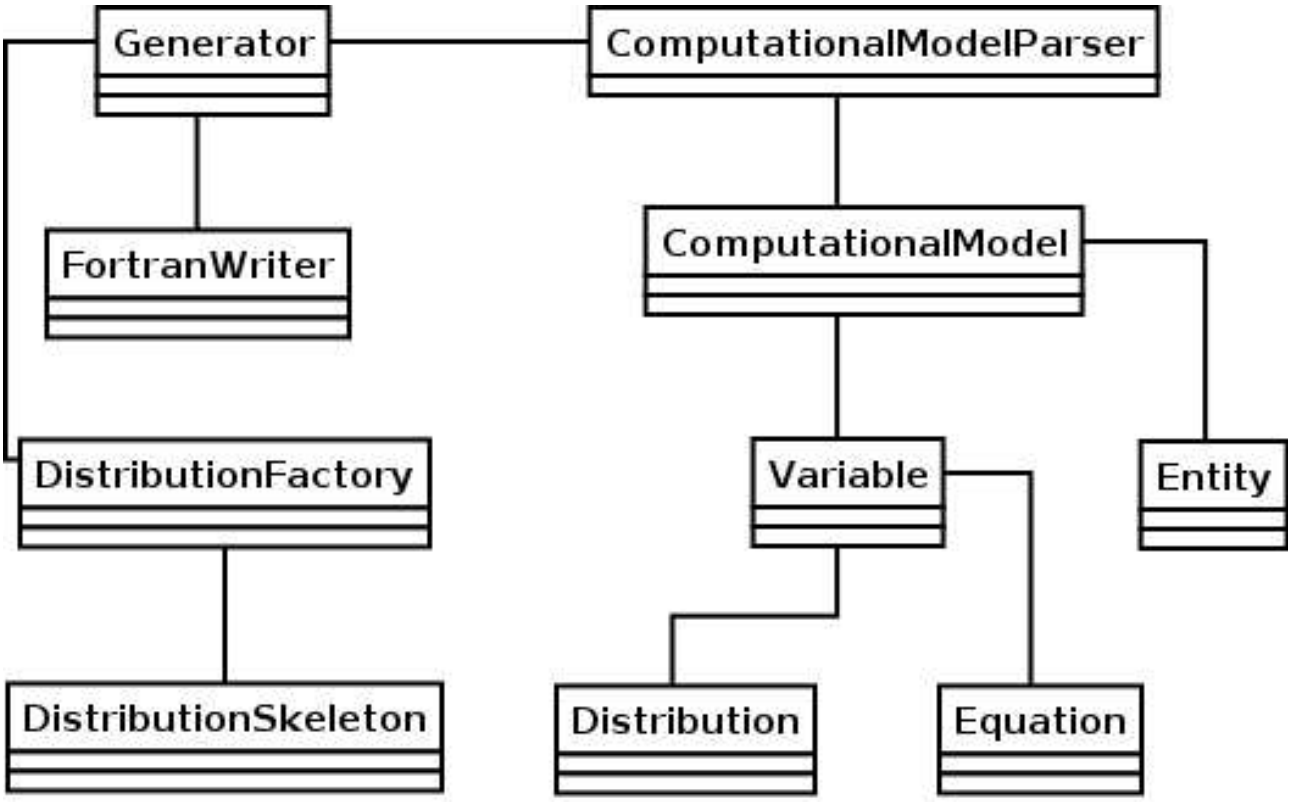

Figure 5: A diagram of the generator class structure.

8.1 Variable

A number of variables are defined in the model file. These variables are represented by Variable objects. One Variable object describes one variable defined in the file. Equations and distributions in the model file define dependencies which are used to link the objects. Some of the fields are filled in after the reading of simulation parameter files, for example the proposal distribution. Distributions are saved in the Variable objects as Distribution objects. Equations are saved in the Variable objects as Equation objects. A Variable object can have either a distribution (stochastic) or an equation (functional). Both options have their own constructors.

8.1.1 Fields of Variable

Field name	Type	Description
name	String	The name of the variable.
belongsTo	Entity	The Entity the variable is associated with, or Null if it isn't associated with any entity.
affects	LinkedList <Variable>	Pointers to each Variable which this Variable affects.
depends	LinkedList <Variable>	Pointers to each Variable of which this Variable depends on.
data	boolean	True if the variable is data, otherwise False.
column	int	The column the variable is found in if the variable is data.
functional	boolean	True if the variable is functional, False if it's stochastical.
equation	Equation	The equation for the variable if the variable is functional, otherwise Null.
distribution	Distribution	The Distribution for the variable if the variable is stochastic, otherwise Null.
proposal	Distribution	The proposal distribution of the variable.
missingValues	int	The number of missing values if the variable is data.
algorithm	String	The algorithm that is used for updating the variable. The program will use only one algorithm, so this field is needed only if someone expands the program to use other algorithms.
proposalStrategy	String	The proposal strategy used for the variable.
typeInteger	boolean	True if the variable is an integer, False if it's a double.
updates	int	The number of minimum updates for the variable.
printed	boolean	True if the variable is printed during the simulation, false otherwise.

8.1.2 Operations of Variable

Operation	Return type	Description
Variable(Entity belongsTo, int column, boolean data, String name, distribution Distribution)	-	Constructor for a Variable object.
Variable(Entity belongsTo, int column, boolean data, String name, Equation equation)	-	Constructor for a Variable object.
Variable()	-	Constructor for a Variable object.
getAffectsList()	LinkedList <Variable>	Returns a list of all Variables which this Variable affects.
addAffected(Variable variable)	void	Adds a Variable to affects list.
getDependenceList()	LinkedList <Variable>	Returns a list of all Variables which depend on this Variable.
addDependence(Variable variable)	void	Adds a Variable to dependence list.
getAlgorithm()	String	Returns the algorithm used to update this Variable.
setAlgorithm(String algorithm)	void	Sets the algorithm used to update this Variable.
getColumn()	int	Returns the column of a data file from which this Variable is found.
setColumn(int column)	void	Sets the column of a data file from which this Variable is found.
isData()	boolean	Returns true if Variable comes from data, false otherwise.
setData(boolean data)	void	Sets the attribute describing whether the Variable comes from data.
getEquation()	Equation	Returns an Equation object if the Variable is functional, null is returned otherwise.
setEquation(Equation equation)	void	Set an Equation object for the Variable is functional.
getDistribution()	Distribution	Returns a Distribution object if the Variable is stochastic, null is returned otherwise.
setDistribution(Distribution distribution)	void	Sets a Distribution object for the Variable.
isFunctional()	boolean	Returns true if the Variable is functional, false otherwise.
setFunctional(boolean)	boolean	Sets the attribute describing whether the Variable is functional.
getProposal()	Distribution	Returns Variable's proposal distribution.
setProposal(Distribution proposal)	void	Sets a proposal distribution to the Variable.
getStrategy()	String	Returns proposal strategy.
setStrategy(String strategy)	void	Sets a proposal strategy.
getUpdates()	int	Returns the number of updates.
setUpdates(int updates)	void	Sets the number of updates.
getMissingValueCount()	int	Returns the count of missing values in data.
incrementMissingValues()	void	Increases missing values by one.

isInteger()	boolean	Returns true if the Variable is an integer, false otherwise.
setType(boolean typeInteger)	void	Sets the type of the Variable object.
getEntity()	Entity	Returns the Entity object the Variable belongs to or null if the variable is global.
setEntity(Entity entity)	void	Sets the Entity object the Variable belongs to.
setPrinted()	void	Sets the Variable to be printed.
isPrinted()	boolean	Returns true if the variable needs to be printed, false otherwise.
isOk()	boolean	Checks that all necessary fields are set.

8.2 Entity

A number of structures are defined in the model file. A structure can link a number of variables, defining the indexing and the data file(s) used. An Entity object is constructed for each structure. Entity objects are used when correct indexing for variables is computed. The data file names are also saved in the objects.

8.2.1 Fields of Entity

Field name	Type	Description
dataFile	String	The name of the file where the data related to the entity is found. Null if the the entity is not related to data.
isMatrix	boolean	True if the data is in matrix format, otherwise false.
name	String	The name of the Entity.
size	int	The number of entities of this type.
spatialMatrixFile	String	The name of the file where the spatial matrix is found. Null if the entity is not spatial.
variableList	LinkedList <Variable>	The list of Variables related to the Entity.
xCoordinateString	String	The name of the Entity that the horizontal dimension in the spatial matrix represents. Null if the entity is not spatial.
yCoordinateString	String	The name of the Entity that the vertical dimension in the spatial matrix represents. Null if the entity is not spatial.
xCoordinate	Entity	The Entity that the horizontal dimension in the spatial matrix represents. Null if the entity is not spatial.
yCoordinate	Entity	The Entity that the vertical dimension in the spatial matrix represents. Null if the Entity is not spatial.

8.2.2 Operations of Entity

Operation	Return type	Description
Entity(String name, String spatialMatrix, String xCoordinateString, String yCoordinateString)	-	Constructor for an Entity object.
Entity()	-	Constructor for an Entity object.
addVariable(Variable variable)	void	Adds a Variable object to the variable list.
getVariableList()	LinkedList <Variable>	Returns the variable list.
setName(String name)	void	Sets the name of the variable.
setSize(int size)	void	Sets the size of the object. This is the number of lines in the data file.
getSize()	int	Returns the size of the object.
getDataFile()	String	Returns the name of the data file.
isMatrix()	boolean	Returns true if the Entity combines two other Entities, and thus is a matrix.
setMatrix(boolean isMatrix)	void	Sets the isMatrix attribute of the entity.
getSpatialMatrixFile()	String	Returns the file name of spatial dependency matrix. Null is returned if there is no matrix.
setSpatialMatrixFile(String spatialMatrixFile)	void	Sets the name of the spatial matrix file.
getXCoordinate()	Entity	Returns the horizontal dimension Entity. Null is returned if the Entity is not a matrix.
setXCoordinate(Entity XCoordinate)	void	Sets the horizontal dimension Entity.
getYCoordinate()	Entity	Returns the vertical dimension Entity. Null is returned if the Entity is not a matrix.
setYCoordinate(Entity YCoordinate)	void	Sets the vertical dimension Entity.
link(HashMap <String, Entity> mapper)	void	Links the Entity object to other Entity objects.

8.3 ComputationalModel

Figure 6: ComputationalModel and its relationships.

The Parser returns a ComputationalModel object to the generator. The object has all the needed information to construct a working Fortran program which computes the given problem.

8.3.1 Fields of ComputationalModel

Field name	Type	Description
iterations	int	The number of total iterations. This attribute is valid only if the update strategy is sequential.
burnIn	int	The number of iterations the program does before the printing of variables starts.
thinning	int	The number of iterations between the printing of variables.
updateStrategy	String	The update strategy used: sequential or random.
variableList	LinkedList <Variable>	A linked list of all global Variable objects.
entityList	LinkedList <Entity>	A linked list of all Entity objects.
entityMapper	HashMap <String, Entity>	A collection which combines Entity objects and their names.
variableMapper	HashMap <String, Variable>	A collection which combines all global Variable objects and their names.
modelFile	String	The file name of the model description file.

8.3.2 Operations of ComputationalModel

Operation	Return type	Description
ComputationalModel(int iterations, int burnIn, int thinning, String updateStrategy, LinkedList <Variable> variableList, LinkedList <Entity> entityList, HashMap <String, Entity> entityMapper, HashMap <String, Variable> variableMapper, String modelFile)	-	Constructor for a ComputationalModel object.
getIterations()	int	Returns the total iterations, if specified. If the update strategy is random, this definition is not appropriate and this method returns -1.
getBurnIn()	int	Returns the length of the burn-in period.
getThinning()	int	Returns the thinning.
getUpdateStrategy()	String	Returns the update strategy.
getVariableList()	LinkedList <Variable>	Returns the linked list of all global Variable objects.
getEntityList()	LinkedList <Entity>	Returns the linked list of all Entity objects.
getEntityMapper()	HashMap <String, Entity>	Returns the HashMap collection which combines all Entity objects and their names.
getVariableMapper()	HashMap <String, Variable>	Returns the HashMap collection which combines all Variable objects and their names.
getModelFile()	String	Returns the name of the model description file.

8.4 Equation

An Equation is constructed for each functional variable. All variables in the functional variable's equation and the equation itself are saved in this Equation object.

8.4.1 Fields of Equation

Field name	Type	Description
parameterString	String[]	An array of all variables in a equation. This field is used to store the variables before the linking.
parameters	Variable[]	An array of all variables in a equation. This field is used to store the variables after the linking.
equation	String	The equation of the object.
startingIndex	int[][]	Matrix of starting positions of all the variables in the equation.

8.4.2 Operations of Equation

Operation	Return type	Description
Equation(String equation, String[] parameterString, int[][] startingIndexMatrix	-	The constructor for Equation.
getEquation()	String	Returns the equation stored in the Equation.
getParameterString()	String[]	Returns an array of parameters as String.
setParameters(Variable[] parameters)	void	Sets the parameters (Variables) used in the equation.
getParameters()	Variable[]	Returns the array of parameters.
getStratingIndexMatrix()	int[][]	Returns the starting index matrix.

8.5 Distribution

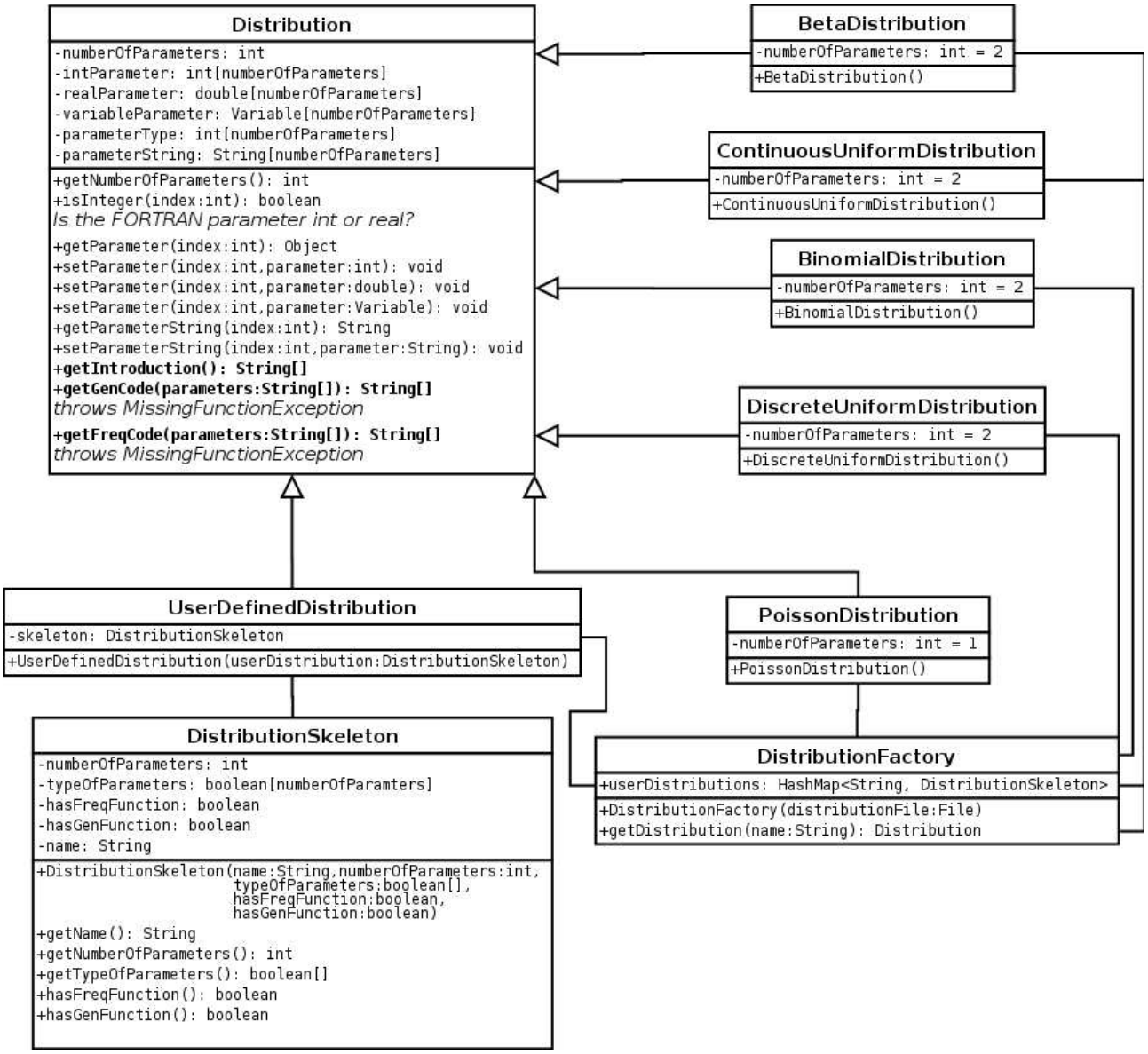

Figure 7: A diagram of the Distribution structure.

The Distribution is an abstract class that provides a simple interface for accessing different distributions' proposal generation and frequency functions without knowing their specifics. Distribution is extended by classes UserDistribution, DiscreteUniformDistribution, BinomialDistribution, PoissonDistribution, ContinuousUniformDistribution and BetaDistribution.

8.5.1 Fields of Distribution

The fields of Distribution are outlined here.

Field name	Type	Description
numberOfParameters	int	The number of parameters for the mathematical function of the distribution.
intParameter	int [numberOfParameters]	Contains the parameters of this Distribution that are fixed integers.
realParameter	double [numberOfParameters]	Contains parameters of this Distribution that are fixed real numbers.
variableParameter	Variable [numberOfParameters]	Stores the parameters that must be referenced from Variable instances.
parameterType	int [numberOfParameters]	Contains a map of the parameter types that is used to index the different type parameter arrays in correct order. Acceptable values are: 0 = integer, 1 = double, 2 = Variable.
parameterString	String [numberOfParameters]	Contains the raw parsed parameter Strings that are used to build the links to the actual parameters according to their names.

8.5.2 Operations of Distribution

This section introduces the operations of Distribution.

Operation	**Return type**	**Description**
getNumberOfParameters()	int	returns the value of *numberOfParameters*.
isInteger(int index)	boolean	Returns *true* if the parameter reference at *index* is to be an INTEGER in the Program to be generated, otherwise returns *false*.
getParameter(int index)	Object	Returns the parameter at *index*.
setParameter(int index, int parameter)	void	sets *parameter* into *index* of *intParameter*, and updates *parameterType* accordingly.
setParameter(int index, double parameter)	void	sets *parameter* into *index* of *realParameter*, and updates *parameterType* accordingly.
setParameter(int index, Variable parameter)	void	sets *parameter* into *index* of *variableParameter*, and updates *parameterType* accordingly.
getParameterString(int index)	String	Returns the parsed parameter String at *index* of *parameterString*
setParameterString(int index, String parameter)	void	Sets the parsed parameter String at *index* of *parameterString*
abstract getIntroduction()	String[]	Returns the introduction lines necessary for the distribution modules. This is an EXTERNAL definition.
abstract getGenCode(String[] parameters	String	Returns the Fortran call for the proposal generation subroutine for the distribution as a String[].
abstract getFreqCode(String[] parameters)	String parameters	Returns the Fortran call for the frequency subroutine for the distribution as a String[].

8.6 DistributionSkeleton

The DistributionSkeleton serves as a collection of information that DistributionFactory uses for constructing UserDefinedDistribition instances.

8.6.1 Fields of DistributionSkeleton

Field name	Type	Description
numberOfParameters	int	The number of parameters for the mathematical function of the distribution.
typeOfParameters	boolean [numberOfParameters]	Contains the types of parameters of this skeleton of a user-defined distribution. The value of an index is *true* if the parameter at the index in question should be an integer, otherwise it is *false*.
hasFreqFunction()	boolean	*true* iff the distribution in question has a frequency function, that is the user distributions file has the header *name*_freq and such a subroutine exists there.
hasGenFunction()	boolean	*true* iff the distribution in question has a proposal generation function, that is the user distributions file has the header *name*_gen and such a subroutine exists there. Note that user generation subroutines are expected to generate an arrayful of proposals on a single invocation.
name	String	Contains the name of the distribution, this being the first part of the distribution's corresponding subroutine names mentioned above.

8.6.2 Operations of DistributionSkeleton

These operations allow for query of field values, only. Setting the field values is always done upon creation, see the constructor.

Operation	Return type	Description
DistributionSkeleton (String name, int numberOfParameters, boolean[] typeOfParameters, boolean hasFreqFunction, boolean hasGenFunction)	-	The constructor: creates a new DistributionSkeleton instance. This will be called after reading the user-defined distributions from the user distribution file, once for each such distribution name.
getName()	String	Returns the value of *name*.
getNumberOfParameters()	int	returns the value of *numberOfParameters*.
getTypeOfParameters()	boolean[]	Returns a reference to *typeOfParameters*
hasFreqFunction()	boolean	Returns the value of *hasFreqFunction*
hasGenFunction()	boolean	Returns the value of *hasGenFunction*

8.7 DistributionFactory

The DistributionFactory stores information about the distributions, both user-defined and provided. It can be used to match a distribution name to its corresponding Distribution entity and create an instance of this for the linking of a Variable to other Variables via its Distribution.

8.7.1 Fields of DistributionFactory

Field name	Type	Description
userDistributions	HashMap <String, DistributionSkeleton>	Contains a map of DistributionSkeletons that can be accessed by the distribution names.

8.7.2 Operations of DistributionFactory

The operations used for acquiring distributions for variables.

Operation	Return type	Description
DistributionFactory (File distributionFile)	-	The constructor: associates the new instance with a given user-defined distributions file.
getDistribution(String name)	Distribution	Returns a reference to a newly created Distribution with the given *name*, by first indexing the *userDistributions* and then constructing a Distribution subclass from the information.

8.8 UserDefinedDistribution

UserDefinedDistribution is a subclass of Distribution. An UserDefinedDistribution object is constructed for each user given distribution.

8.8.1 Fields of UserDefinedDistribution

UserDefinedDistribution has the same fields as other Distribution class' subclasses have. It also has a field for a DistributionSkeleton object.

8.8.2 Operations of UserDefinedDistribution

The UserDefinedDistribution represents a non-standard distribution instance. It differs from Distribution only with its constructor.

Operation	**Return type**	**Description**
UserDefinedDistribution (DistributionSkeleton userDistribution)	-	The constructor: creates a new instance according to the information in *userDistribution*.

9 Modules

This section describes the modules of the Generator, that is the Java classes that are used to generate the Program to simulate the model in question. The classes are outlined with their interfaces outside and their summarized internal functionality.

9.1 ComputationalModelParser

The parser is the part of the generator that reads the model and simulation input files, puts the data into correct places and returns the data structure to the main generator program.

9.1.1 Interface

Operation	**Return type**	**Description**
readModel(String modelFileName, String initialValueFileName, String simulationFileName, String proposalFileName, String updateFileName, String toOutputFileName, DistributionFactory factory) throws IOException, SyntaxException, MissingFunctionException	ComputationalModel	Parses all the files and constructs a ComputationalModel instance with Variables linked to their Entities, Entities to each other and Equations and Distributions set for variables.

9.1.2 Internal operations

Operation	Return type	Description
readInitialValues(File file, HashMap <String, Variable> variableMapper, File missingFile)	void	Reads the initial values for the variables from *file* and assigns them to the correct Variables. Incorrect file format will cause a *SyntaxException* to be thrown, missing initial values will cause an error message and stop the program execution.
readData(File file, Variable datavar, File missingFile)	void	Reads the data for a single variable and sets the fields: datavar.missingValues, datavar.belongsTo.size. Reads out the missing values and writes the datafile_missing.txt and creates *missingFile* to be used by later methods (readInitialValues). Invalid symbols or uneven line lengths in the data will generate a *SyntaxException*.
readProposal(File file, HashMap <String, Variable> variableMapper)	void	Reads proposal distributions from *file* and assigns them for the correct Variables. Missing distributions will cause an error message and stop execution.
readUpdate(File file, String updateStrategy, int iterations)	void	Reads the file, parses the update strategy into *updateStrategy* and iterations into *iterations*. If the update strategy is random, parses each iterations value into the correct Variable's updates field.
readSimulation(File file, int burnIn, int thinning)	void	Reads values from the file into burnIn and thinning.
readOutput(File file, HashMap <String, Variable> variableMapper)	void	Reads the variables to be output from the file sets variable.setPrinted() for those variables.

9.1.3 Fields filled in by the parser

9.1.3.1 ComputationalModel

Field name	type
iterations	int
burnIn	int
thinning	int
updateStrategy	String
variableList	LinkedList <Variable>
entityList	LinkedList <Entity>
variableMapper	HashMap <String, Variable>
entityMapper	HashMap <String, Entity>

9.1.3.2 **Entity**

Field name	Type
data	String
isMatrix	boolean
name	String
size	int
spatialMatrix	String
variableList	LinkedList <Variable>
xCoordinateString	String
yCoordinateString	String

9.1.3.3 **Variable**

Field name	Type
name	String
belongsTo	Entity
data	boolean
column	int
functional	boolean
equation	Equation
distribution	Distribution
proposal	Distribution
missingValues	int
algorithm	String
proposalStrategy	String
typeInteger	boolean
updates	int

9.2 FortranWriter

FortranWriter receives lines of Fortran source code and writes them to a file correctly indented and wrapped to 79 character length.

9.2.1 Interface

Operation	Return type	Description
FortranWriter(String fileName)	-	Constructor, specifies which file to write
write(String line) throws IOException	void	Writes *line* to the specified file, correctly indented and multilined if longer than 79 characters
write(String[] lines) throws IOException		Writes *lines* to the specified file, correctly indented and multilined if longer than 79 characters

9.2.2 Indentation

If one of the following keywords is found on a line, the next line begins an indented section.

- BLOCK DATA
- DO
- FORALL
- FUNCTION
- IF
- INTERFACE
- MODULE
- PROGRAM
- SELECT CASE
- SUBROUTINE
- TYPE typename
- WHERE

If END is found on a line, the previous line was the last line of an indented section.

Keywords that mark both the ending of the previous indented section and the beginning of another:

- CASE
- CONTAINS
- ELSE
- ELSEWHERE

9.2.3 Line wrapping

If a line is over 79 characters long, it's cut at the last whitespace found so that it's no longer than 77 characters. ' &' is added to the end of the line and the rest of it is moved to the next line, indented. If the remaining line is too long as well, it receives the same treatment excluding the indenting of the following line.

10 Algorithms

This section describes how the generator data structures can be used when generating the Fortran program.

10.1 Linking

When the Parser has read the model, it is necessary to link the variables to each other.

The attributes set in this phase are:

- In *Entity* objects: the *XCoordinate* and *YCoordinate*
- In *Variable* objects: the list *depends* includes the variables the current variable depends on, that is, the variables that affect the current variable
- In *Variable* objects: the list *affects* includes the variables the current variable affects, that is, the variables that depend on the current variable
- In *Distribution* objects: the reference parameters of the distribution, that is, the array *variableParameter*
- In *Equation* objects: the parameters of the equation, that is, the array *parameters*.

Note: We assume that the Parser creates the (unlinked) *Distribution* objects for all the stochastic variables. The objects store the names of its parameters as *Strings*. Similarly we assume that the Parser creates the (unlinked) *Equation* objects for all the functional parameters. The objects store as *Strings* the names of variables appearing in the equation.

The idea of having this kind of information is that when generating the acceptance probability calculation for a parameter, the *affects* and *depends* lists define which variables are part of the probability calculation formula. By using the *Distribution* object we obtain parameters needed when the distribution is used in the acceptance probability formula.

When the variable is stochastic, the *Distribution* object already has information of the variables which affect the current variable, so in that case the *depends* contains redundant information but it is used anyway. When the variable is functional, the *Equation* objects contains the same information.

The Parser has created the *Variable* and *Entity* instances which correspond to the variables in the model. In this phase no objects need to be created.

10.1.1 Linking the Entity objects

The linking of the Entity objects is done in the Parser.

- Put all *Entity* objects into a *HashMap*. The key of an *Entity* is its *name*.

- Iterate the entityList (the data structure of the model where all the Entities are) to find the entities of the model
- For each *Entity* found:
  - If the entity doesn't describe a matrix, do nothing.
  - Otherwise check the attributes *XCoordinateString* and *YCoordinateString* that define the corresponding parent entities (x-coordinate and y-coordinate entities), find the corresponding *Entity* objects from the *HashMap*, and link them to attributes *XCoordinate* and *YCoordinate*.

10.1.2 Linking the Variable, Distribution and Equation objects

A *Distribution* object has its parameters as *Strings*. An *Equation* object has the variable names appearing in it as *Strings*.

- Put all *Variable* objects into a *HashMap*. The key of a variable is its *name*.
- Iterate the *VariableList* in *ComputationalModel* and the *VariableList* objects in all the *Entities* to find all the variables of the model. For each one the attribute *functional* tells whether the variable is stochastic or functional.
- For each stochastic variable x:
  - Get the distribution of the variable as a *String*
  - For each variable name "y" that occurs in the distribution:
  - Use the *HashMap* to find out the corresponding *Variable* object y
  - Add y to the distribution parameter data structure
  - Add x to y's affects-list
  - Add y to x's depends-list
- For each functional parameter x:
  - Get the equation of the parameter
  - For each variable name "y" that occurs in the equation:
  - Use the *HashMap* to find out the corresponding *Variable* object y
  - Add y to the equation variables data structure
  - Add x to y's affects-list
  - Add y to x's depends-list

10.2 Generating the acceptance probability calculation code

This chapter describes how the generator data structures are used when generating the Fortran code which calculates the acceptance probability for a single parameter.

The following connections between the objects are needed:

- The variables[1] on which the current parameter depends
- The distribution which describes the dependency (the name of the distribution and the order of its parameters)
- The variables which depend on the current parameter
- The distributions which describe the dependencies (the names of the distributions and the order of their parameters)
- The proposal distribution and the proposal strategy of the current parameter

When these fields are correctly set, it is possible to generate the code which calculates the acceptance probability.

The basic algorithm for generating the probability calculation code for one variable is:

- Initialize p_acc (an internal variable in the update subroutine describing the acceptance probability of the variable being updated) to 1 and generate a proposal
- Considering the current parameter's distribution...
- generate a subroutine call: the frequency function[2] with the parameters defined and **the old value** (store the result into a temporary variable)
- generate a subroutine call: the frequency function with the parameters defined and **the new value** (store the result into a temporary variable)
- Generate code which divides the latter by the former and multiplies p_acc with it
- Loop through the variables which depend on the current parameter. For each:
  - Based on its distribution...
  - generate a subroutine call: the frequency function with its parameters including **the old value of the current variable** and the value of the depending variable (store the result into a temporary variable)
  - generate a subroutine call: the frequency function with its parameters including **the new value of the current variable** and the value of the depending variable (store the result into a temporary variable)

[1] In this section, the term variable is used when referring to a variable or a parameter and when it's irrelevant which one there really is.

[2] Note that frequency function is used as a mathematical term whereas the result is often calculated by using a (Fortran) subroutine, not a (Fortran) function.

  - Generate code which divides the latter by the former and multiplies p_acc with it

- If the proposal strategy is random walk:
  - Generate code which calculates the difference “new value - current value”
  - Generate a subroutine call: the density function of the proposal distribution with its parameters and **the opposite number of the difference**
  - Generate a subroutine call: the density function of the proposal distribution with its parameters and **the difference**
  - Generate code which divides the former result by the latter result and multiplies p_acc with it

- If the proposal strategy is fixed proposal distribution
  - Generate a subroutine call: the density function of the proposal distribution with its parameters and **the new value**
  - Generate a subroutine call: the density function of the proposal distribution again with its parameters and **the old value**
  - Generate code which divides the former result by the latter result and multiplies p_acc with it.

  Note: The current value doesn’t affect the next proposed value in any way.

When generating the detailed code for getting the correct values, the following information is needed:

- The entity to which the current parameter belongs
- The entities to which the affected variables belong
- The entities to which the affecting variables belong

For each entity the following information is needed:

- The dimension and the size of the entity
- If the entity describes a intersection of two entities, the parent entities (XCoordinate and YCoordinate)

10.2.1 Indexing

We have the current variable with which we are dealing right now, and which we know how to index. (For example a_i_j, and we know we have to use a % two_dim(i, j) % value.) Then we have another variable, which appears in the distribution of the current variable, but which we don’t know how to index. The other variable may be global,

belonging to the same entity as the current variable or belonging to a different entity as the current variable.

The following cases are possible:

1. The two variables are both global
2. The two variables belong to the same one-dimensional entity
3. The two variables belong to the same two-dimensional entity
4. The affecting variable is global and the current variable belongs to an one-dimensional entity
5. The affecting variable belongs to an one-dimensional entity and the current variable belongs to a two-dimensional entity
6. The affecting variable is global and the current variable belongs to a two-dimensional entity (Note that this can only be done through a functional parameter between them.)

By using *Entity* objects it's possible to find out which case is present.

This is how the previous situations are solved:

1. No special indexing is needed.
2. The same indexing is used for the two variables. For example if we know how to index a % one_dim(i), we index b % one_dim(i) accordingly.
3. Same as the previous case except that we need two indices, for example a % two_dim(i, j) and b % two_dim(i, j).
4. No indexing is needed for the global variable and we already know how to index the current variable.
5. We must use the correct indexing based on the entities of the variables. For example: we have a % two_dim(i, j) and an one-dimensional variable b affects it. We must decide whether we use b % one_dim(i) or b % one_dim(j). If b:s Entity equals the XCoordinate of a, we use b % one_dim(j) and if b:s Entity equals the YCoordinate of a, we use b % one_dim(i). Note that in Fortran the first coordinate is y, not x.
6. No special indexing is needed.

Note that in the generated Fortran program a global variable is stored as a one-dimensional variable with only one unit.

So that's how we can generate code which gets the values of the needed variables. New values are acquired the same way except we use % new_value instead of % value, and we need to keep track which is the variable for which we must use the new value.

10.2.2 Loops

When we are generating the probability calculation code for a current variable, we might need loops, if the current variable affects other variables that belong to a different entity which has more dimensions than the entity of the current variable.

We assume that the current variable is indexed with % one_dim(1) (if it is global), % one_dim(i) (if it is one-dimensional) or % two_dim(i, j) (if it is two-dimensional).

The following cases are possible:

1. The affecting variable and the affected variable are both global
2. The affecting variable and the affected variable belong to the same one-dimensional entity
3. The affecting variable and the affected variable belong to the same two-dimensional entity
4. The affecting variable is global and the affected variable belongs to an one-dimensional entity
5. The affecting variable belongs to an one-dimensional entity and the affected variable belongs to a two-dimensional entity
6. The affecting variable is global and the affected variable belongs to a two-dimensional entity (Note that this can only be done through a functional parameter between them.)

This is how the previous situations are solved:

1. No loops are needed.
2. No loops are needed.
3. No loops are needed.
4. We need to loop through all the variable instances of the affected variable. For example if the global variable a affects all variable b:s related to birds, we must loop through all the b:s with:

```
DO i=1, 200
(where 200 is the size of the entity,
for example the number of the birds)
calculate everything, the indexing is:
a % one_dim(1) (global variable)
b % one_dim(i)
END DO
```

5. We need to loop through the variable instances which belong to the two-dimensional entity, and we need to know which is the current index of the one-dimensional variable. For example if we have the variable **a** related to birds and the variable **b** related to the intersection of birds and squares, we need to loop through all the b:s related to the specific bird:

```
DO j=1, 300
(where 300 is the size of the entity,
for example the number of the squares)
calculate everything, the indexing needed is:
a % one_dim(i)
b % two_dim(i, j) (if a's Entity is b:s YCoordinate)
or b % two_dim(j, i) (if a's Entity is b:s XCoordinate)
END DO
```

6. We need to loop through the variable instances which belong to the two-dimensional entity. For example if we have the variable **a** which is global and the variable **b** related to the intersection of birds and squares, we need to loop through all the b:s related to the specific bird:

```
DO j=1, 300
(where 300 is the size of the entity,
for example the number of the squares)
calculate everything, the indexing needed is:
a % one_dim(1) (global)
b % two_dim(i, j)
END DO
```

10.2.3 Functional parameters

How the functional parameters are dealt with:

- When a parameter is updated, the functional parameters affected by it are also updated.
- When a new value is generated for a parameter, new values are also calculated for all the functional parameters which depend on the parameter.
- A variable is considered to affect another variable also when there is a functional parameter between them.

All needed information related to functional parameters is stored in the *Equation* object.

10.2.4 Generating the function/subroutine calls

Generating the function / subroutine calls is straightforward, when we know the parameters as *Strings*. We simply call the method getFreqCode in the correct Distribution object, passing the wanted parameters as Strings.

For example:
getFreqCode(“alpha % one_dim(i)”, “beta % one_dim(j)”, “x % two_dim(i, j)”, “frequency”)

The method generates the corresponding NAG subroutine call, which calls the frequency function with given parameters and stores the result into the given variable.

11 Correspondence between requirements and design

This chapter describes the correspondence between requirements found in SRS document and design decision found in this document.

Requirement defines the identification and name of the requirement. *Priority* defines the priority of the requirement (E = essential, C = conditional, O = optional), *Design status* describes whether the requirement is designed and going to be implemented and *Chapters* list the chapters of this document related to the particular requirement.

Possible design statuses include:

- Designed: The requirement is going to be supported
- Designed, found in prototype: The requirement is going to be supported and the prototype already supports it
- Not going to be implemented: The requirement is not going to be supported

11.1 Model requirements

Requirement	**Priority**	**Design status**	**Chapters**
M1: Using models	E	Designed	4.1
M2: Defining variables whose values are taken from data	E	Designed	4.1
M3: Defining parameters whose values are not taken from data	E	Designed	4.1
M4: Defining dependencies	E	Designed	4.1
M5: Equations	E	Designed	4.1
M6: Defining variable/parameter repetition structures	E	Designed	4.1
M7: Defining spatial relations	E	Designed	4.1
M9: Reading models from text files	E	Designed	4.1
M10: The distributions used	E	Only distributions found in the NAG library are supported.	5
M11: Distributions defined by the user	C	Designed	4.3, 8.8
M12: Defining distributions	E	Designed	4.1

11.2 Data requirements

Requirement	Priority	Design status	Chapters
D1: The general data format	E	Designed	
D2: Data not available	E	Designed	9.1
D3: Invalid data	E	Designed	9.1

11.3 Simulation requirements

Requirement	Priority	Design status	Chapters
S1: The algorithm used	E	Designed, found in prototype	8.1, 6.1
S2: Choice of algorithm	C	Not going to be implemented	8.1
S3: Setting the number of updates	E	Designed (except when related to blocks), found in prototype	8.1, 6.1
S4: Setting the number of burn-in iterations	E	Designed, found in prototype	4.2.1
S5: Setting the thinning factor	E	Designed, found in prototype	4.2.1
S6: Setting the blocks	C	Not going to be implemented	-
S7: Setting the update strategy	C	Designed, found in prototype	8.1
S8: Setting the weight of the blocks	O	Not going to be implemented	-
S9: Setting the proposal strategies for variables	E	Designed, found in prototype	6.1
S10: Proposal distributions	E	Designed, found in prototype	4.2.3, 6.2.1
S11: Setting initial values	E	Designed, found in prototype	4.2.2, 6.2.2
S12: Defining parameters to output	E	Designed	4.2.5, 6.2.3
S14: Informing the user about the progress	E	Designed	
S15: Soft stop	O	Not going to be implemented	
S16: Parameters in random walk	O	Not going to be implemented	

11.4 Output requirements

Requirement	Priority	Design status	Chapters
OP1: Writing output into a file	E	Designed, found in prototype	6.2.3
OP2: Output file names	E	Designed	4.2.5
OP3: The output	E	Designed, found in prototype	6.2.3
OP4: Information written to output files	E	Designed	
OP5: Summary of the simulation	C	Designed	6.2.3
OP6: File access check	C	Not going to be implemented	-

11.5 General error conditions

Requirement	Priority	Design status	Chapters
E1: File not found	E	Designed, found in prototype	6.2.2
E2: Reporting syntax errors	O	Designed	9.1
E3: Reporting semantic errors	O	Not going to be implemented	-

11.6 Non-functional requirements

Requirement	Priority	Design status	Chapters
N1: Working on Linux	E	Designed, found in prototype	-
N2: The implementation language	E	Designed, found in prototype	6
N3: Parallel computation	O	Not going to be implemented	-
N4: Graphical user interface	O	Not going to be implemented	-

11.7 General requirements

Requirement	Priority	Design status	Chapters
G1: Adding comments to definition files	E	Designed	4

12 References

Rand *Definition of Random Fields in Encyclopedia*
http://encyclopedia.laborlawtalk.com/Random_fields

Tode04 Pekka Tuominen: Todennäköisyyslaskenta I

Math05 http://mathworld.wolfram.com/