PIANOS implementation 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 Introduction

This is the implementation document of the PIANOS project.

This document describes the software after the implementation. The designed features that were changed during the implementation are also described. Suggestions for features of future development are also overviewed in this document.

1.1 Chapters

1. Introduction	Describes this document's purpose.
2. Glossary	Explains the glossary used in this document.
3. Overview of the software	Describes the software and the division into a generator and a generated program.
4. The generated program	Describes the generated program and its modules.
5. Generator	Describes the generator, its packages and classes.
6. Correspondence to requirements	Describes which requirements are implemented.
7. Future development	Overviews future development ideas and features.

1.2 Version history

Version	Date	Modifications
0.2	29.07.2005	Document template
1.0	30.08.2005	First full-featured draft
1.1	31.08.2005	Reviewed and corrected final

2 Glossary

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 candidate obtained from the proposal distribution.

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).

Frequency function is used to refer to a frequency function or a density function that gauges the ‘goodness’ of variable values.

Variable is used to refer to a data variable or an updated parameter. and the Variable class that represents these in the Generator.

Equation: refers to equation of functional variables. The equation is represented by an Equation object.

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.

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.

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.

Real number: The data type for floating point numbers is called ‘real’ in Fortran.

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

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.

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.

Update strategy: The update strategy describes how and in which way variables are updated. Only one update strategy, ‘sequential update’, has been implemented. Random update strategy was designed but now implemented.

Semi-Bayesian model: A Bayesian model which allows cycles when dealing with spatial dependencies.

3 Overview of the software

The software product analyzes spatial data by using a semi-Bayesian model and prior distributions to calculate points from the posterior distributions for a set of variables. This kind of computation is heavy and time consuming on large sets of data. In order to avoid creating one static program, which can run simulations with all possible models, the program consists of two parts: the generator and the generated program. The generator reads the input files and creates a customized Fortran program for the given problem.

The input files define the problem and how the program should run the simulation with the given model. The main input file is the model file, which describes a semi-Bayesian model. The model could describe, for example, a bird species' distribution in Finland. The main topic of interest is the question about the most accurate model to describe a given real life phenomenon. The only way to compare the models is to compare the results. The program will not analyze how accurate the given model is, it will only calculate points from the posterior distribution for variables and print them as output.

The program is limited to small set of possible simulation problems. There are limitations to the models and distributions that can be used. The key feature is the possibility to use spatial relations in the calculations.

The generator can be executed on any computer with the Java runtime environment version 1.5 (or greater) installed. The generated program however needs a Fortran 90 compiler. It also uses NAG libraries, so a computer with a valid NAG license is required. The program was designed with NAG Mark 19.

Use of random update strategy was designed and specified in the design document, but it was not implemented due to schedule pressure. Some other features were dropped earlier in the project. These include, for example, variable blocks and the use of the Gibbs algorithm. Data structures and the generator changed only a little from the specification in the design document. The generator's classes and changes are specified in chapter 5. See chapter 7 for further development ideas.

4 The 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.

4.1 Data structures

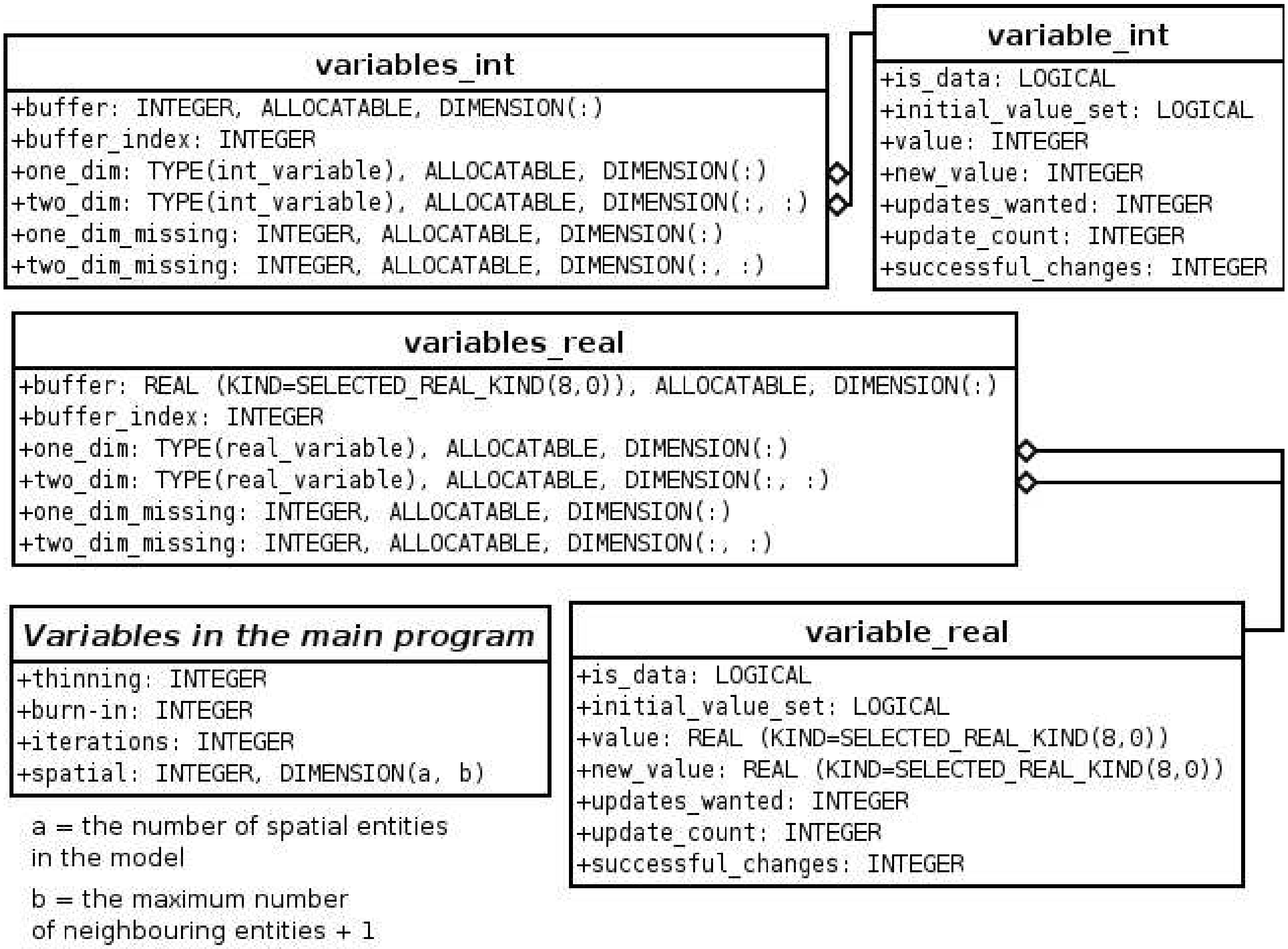

Figure 1: A diagram of the generated data structures.

Figure 1 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. (It is partially implemented, see chapter 7)

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.

4.2 Modules

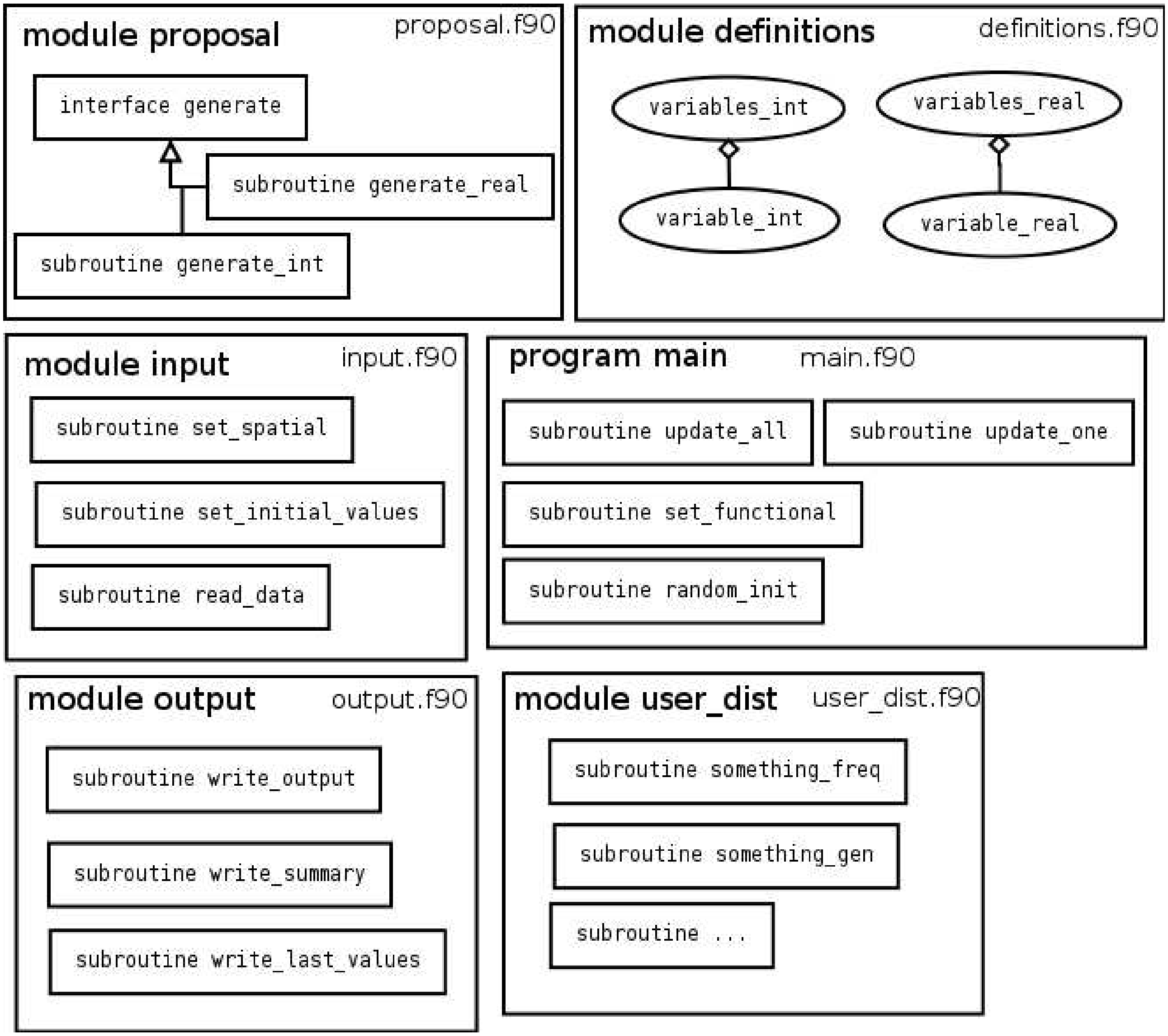

Figure 2: A diagram of the generated modules and their subroutines.

Figure 2 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.

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.

4.2.1 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.

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.

4.2.2 Module input

Subroutine name:	read_data
Description:	Reads the data from data files into the data structure defined in figure 1.
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.

Subroutine name:	set_initial_values
Description:	This subroutine reads the initial values from a data file into the data structure defined in figure 1.
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.

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.

4.2.3 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

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

4.2.4 Program main

Program name: main

Description: This is the main program which performs the simulation by using the subroutines described below.

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, 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 updated as the next iteration is performed.

Subroutine name:	set_functional
Description:	This subroutine sets the values of functional variables before the simulation.
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 of functional variables are updated.

5 The generator program

The software consists of the following packages and classes:

PIANOS.datastructures

- BetaDistribution
- BinomialDistribution
- ComputationalModel
- ContinuousUniformDistribution
- DiscreteUniformDistribution
- Distribution
- DistributionFactory
- DistributionSkeleton
- Entity
- Equation
- PoissonDistribution
- UserDefinedDistribution
- Variable

PIANOS.exceptions

- EntityNotFoundException
- IllegalParametersException
- InvalidModelException
- InvalidProposalException
- MissingDistributionException
- MissingFunctionException
- SyntaxException

PIANOS.generator

- Acceptation
- Definitions
- FortranMain
- Input
- Output
- Proposal

PIANOS.io

- ComputationalModelParser
- FortranWriter

PIANOS

- Generator

5.1 Generator

The Generator class includes the main() and writeProgram() methods. writeProgram() can be used for example a graphical user interface. It has the same functionality as main(). The Generator uses the Parser to read the input files. After it the Generator uses the classes in the package generator. Those classes generate the different Fortran modules. The main() method of the class Generator expects the user to give the input files as command line parameters. The order of these files must be:

1. User defined distribution file
2. Model file
3. Initial value file
4. Simulation parameter file
5. Proposal distribution file
6. Update strategy file
7. The file defining which variables are output
8. The file to save the last values to

Input file names can also be given in a single file that is given to the Generator, See the PIANOS manual for details.

5.2 package PIANOS.datastructures

This package contains the data structures of the software.

5.2.1 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).

5.2.1.1 **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 that this Variable affects.
depends	LinkedList <Variable>	Pointers to each Variable that 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 stochastic.
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. This field is not used since the random update strategy is not implemented.
printed	boolean	*true* if the variable is printed during the simulation, *false* otherwise.
spatial	boolean	*true* if the variable is a functional variable with a spatial expression (SUM or COUNT), *false* otherwise.

5.2.1.2 **Operations of Variable**

Operation	**Return type**	**Description**
Variable()	-	Constructor for a Variable object.
getAffectsList()	LinkedList <Variable>	Returns a list of all Variables that this Variable affects.
addAffected(Variable variable)	void	Adds a Variable to *affects*.
getDependsList()	LinkedList <Variable>	Returns a list of all Variables that depend on this Variable.
addDependence(Variable variable)	void	Adds a Variable to *depends*.
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 in which this Variable is found.
setColumn(int column)	void	Sets the column of a data file in which this Variable is found.
isData()	boolean	Returns *true* if this Variable comes from data, *false* otherwise.
setData(boolean data)	void	Sets whether the Variable comes from data.
getEquation()	Equation	Returns the Variable's Equation if the Variable is functional, *null* is returned otherwise.
setEquation(Equation equation)	void	Set the Equation for the Variable.
getDistribution()	Distribution	Returns a Distribution object if the Variable is stochastic, *null* is returned otherwise.
setDistribution(Distribution distribution)	void	Sets the Distribution for the Variable.
isFunctional()	boolean	Returns *true* if the Variable is functional, *false* otherwise.
setFunctional(boolean funct)	void	Sets whether the Variable is functional.
isSpatial()	boolean	Returns *true* if the Variable is spatial, *false* otherwise.
setSpatial(boolean spatial)	void	Sets whether the Variable is spatial (functional and has a SUM(&x) -style equation).
getName()	String	Returns the name of the variable.
setName(String)	void	Sets the name of the variable.
getProposal()	Distribution	Returns the variable's proposal distribution.
setProposal(Distribution proposal)	void	Sets the proposal distribution to the Variable.

getStrategy()	String	Returns the variable's proposal strategy.
setStrategy(String strategy)	void	Sets the 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.
getEntity()	Entity	Returns the Entity the Variable belongs to or *null* if the variable is global.
setEntity(Entity entity)	void	Sets the Entity that the Variable belongs to.
setPrinted()	void	Sets the Variable to be printed during simulation.
isPrinted()	boolean	Returns *true* if the variable is printed during the simulation, *false* otherwise.
isOk()	boolean	Checks that all necessary fields are set and that dependencies are sane.

5.2.2 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.

5.2.2.1 **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 adjacency matrix is found. *null* if the entity is not spatial.
variableList	LinkedList <Variable>	The list of Variables related to the Entity.
xCoordinate	Entity	The Entity that the horizontal dimension in the data matrix represents. *null* if the entity is not an intersection of two entities.
yCoordinate	Entity	The Entity that the vertical dimension in the data matrix represents. *null* if the Entity is not an intersection of two entities.

5.2.2.2 **Operations of Entity**

Operation	**Return type**	**Description**
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 entity.
getName()	String	Returns the name of the entity.
setSize(int size)	void	Sets the size of this Entity. This is the number of lines in the data file.
getSize()	int	Returns the size of the object.
setLineLength(int lineLength)	void	Sets the line length.
getLineLength()	int	Returns the line length.
getDataFile()	String	Returns the name of the data file.
setDataFile(String dataFile)	void	Sets 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 the adjacency matrix. *null* is returned if there is no matrix.
setSpatialMatrixFile(String spatialMatrixFile)	void	Sets the name of the adjacency 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.
isSpatial()	boolean	Returns *true* if the entity is spatial.

5.2.3 ComputationalModel

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.

5.2.3.1 **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'. The random update strategy is not implemented
maxSpatialNeighbours	int	The maximum number of spatial neighbours in the simulation.
modelFile	String	The file name of the model description file.
initialValueFile	String	The file name of the initial value file.
outputFile	String	The file name of the output file.
summaryFile	String	The file name of the summary file.
lastValuesFileName	String	The file name of the last values file.
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 Variables and their names.
topologicalVariableList	ArrayList <Variable>	A collection of all variables in topological order.

5.2.3.2 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.
getNeighbourCount()	int	Returns the maximum number of spatial neighbours.
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.
getModelFileName()	String	Returns the name of the model description file.
getOutputFileName()	String	Returns the name of the output file.
getInitialFileName()	String	Returns the name of the initial value file.
getSummaryFileName()	String	Returns the name of the summary file.
getLastValuesFileName()	String	Returns the name of the last values file.
getTopologicalVariableList()	ArrayList <Variable>	Returns a topologically ordered collection of Variable objects.

5.2.4 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.

5.2.4.1 **Fields of Equation**

Field name	**Type**	**Description**
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, broken down so that every index is either a single variable name or something else (so that each index can be matched against variable names and replaced with the final Fortran expression)

5.2.4.2 **Operations of Equation**

Operation	**Return type**	**Description**
Equation(String[] equation, Variable[] parameters)	-	The constructor for Equation.
getEquation()	String[]	Returns the equation stored in the Equation.
setParameters(Variable[] parameters)	void	Sets the parameters (Variables) used in the equation.
getParameters()	Variable[]	Returns the array of parameters.
setParameters(Variable parameter)	void	Sets *Parameters* to be $\{parameter\}$. This is useful for Equations with spatial expressions.

5.2.5 Distribution

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 UserDefinedDistribution, DiscreteUniformDistribution, BinomialDistribution, PoissonDistribution, ContinuousUniformDistribution and BetaDistribution.

5.2.5.1 **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.

5.2.5.2 **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()	ArrayList<String>	Returns the introduction lines (EXTERNAL) necessary for using this distribution.
abstract getGenCode(String[] parameters	ArrayList<String>	Returns the Fortran call for the proposal generation subroutine for the distribution as a String[].
abstract getFreqCode(String[] parameters)	ArrayList<String>	Returns the Fortran call for the frequency subroutine for the distribution as a String[].

5.2.6 DistributionSkeleton

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

5.2.6.1 **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*.
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.

5.2.6.2 **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*
hasGenFunction()	boolean	Returns the value of *hasGenFunction*

5.2.7 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.

5.2.8 Fields of DistributionFactory

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

5.2.8.1 **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. Returns also internal Distributions such as BetaDistribution.

5.2.9 UserDefinedDistribution

UserDefinedDistribution is a subclass of Distribution. A UserDefinedDistribution object is constructed for each user-defined distribution used in the model.

5.2.9.1 **Fields of UserDefinedDistribution**

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

5.2.9.2 **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*.

5.3 package PIANOS.io

This package contains the classes that read the input or write the output.

5.3.1 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.

5.3.1.1 **Operations**

Operation	**Return type**	**Description**
readModel(String modelFileName, String initialValueFileName, String simulationFileName, String proposalFileName, String updateFileName, String toOutputFileName, DistributionFactory factory) throws IOException, SyntaxException, MissingDistributionException	Computational-Model	Parses all the files and constructs a ComputationalModel instance that represents the model files given to the Parser. Calls other methods of the Parser. Called by the Generator.
readModelFile(File file, LinkedList <Variable> variableList, LinkedList <Entity> entityList, HashMap <String, Entity> entityMapper, HashMap <String, Variable> variableMapper, DistributionFactory fact) throws IOException, SyntaxException, MissingDistributionException	void	Reads the model file and puts the information into the maps and lists provided.
parseVariable(File file, boolean isInteger, String toParse, HashMap <String, Variable> variableMapper, DistributionFactory fact) throws SyntaxException, MissingDistributionException	Variable	Reads a variable definition line and returns a representation of the variable, public for testing purposes.
readEntity(File file, LinkedList <Entity> entityList, HashMap <String, Entity> entityMapper, HashMap <String, Variable> variableMapper, String header, Scanner reader, DistributionFactory fact) throws SyntaxException, MissingDistributionException	void	Reads an entity section in the model file and saves the information into the structures provided.

readSimulation(File file) throws SyntaxException, IOException	int[]	Reads a simulation parameter file that gives burn-in and thinning. Returns {burn-in, thinning}.
readUpdate(File file, HashMap <String, Variable> variableMapper, String updateStrategy) throws SyntaxException, IOException	Object[]	Reads the update strategy file, saves individual updates (if any) into the variables in the maps, returns {String updateStrategy, int iterations}
readOutput(File file, HashMap <String, Variable> variableMapper) throws SyntaxException, IOException	String[]	Reads the parameter names to output -file and sets isPrinted-flags.
readProposal(File file, HashMap <String, Variable> variableMapper, DistributionFactory fact) throws SyntaxException, IOException, MissingDistributionException	void	Reads the proposal distributions/strategies file, sets them for specified variables.
readEntityData(LinkedList <Entity> entityList) throws IOException, SyntaxException	void	Reads through entities' data files to see if they are correct. builds the missing values matrix files.
readInitialValues(File file, HashMap <String, Variable> variableMapper) throws IOException, SyntaxException	void	Reads the initial values definition file and checks that the format is correct and all missing values of data are present.
countNeighbours(LinkedList <Entity> entityList) throws SyntaxException, FileNotFoundException	int	Counts the maximum number of spatial neighbours in the model.

5.3.2 FortranWriter

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

5.3.2.1 **Interface**

Operation	Return type	Description
FortranWriter(String fileName)	-	Constructor, specifies which file to write
write(ArrayList <String> lines) throws IOException		Writes *lines* 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

5.3.2.2 **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

5.3.2.3 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 line is cut from the middle of a character string literal, ‘&’ is appended to the beginning of the cut, too. If the remaining line is too long as well, it receives the same treatment excluding the indenting of the following line.

5.4 package PIANOS.generator

This package contains classes that generate different parts of the Fortran program.

5.4.1 FortranMain

This class generates the Fortran module main (see 4.2.4).

Public methods

Operation	**Return type**	**Description**
generateMain (String callParameters, ComputationalModel model) throws IOException, InvalidModelException, IllegalParametersException, MissingFunctionException	void	Generates and writes the Fortran module main.f90.

Private methods

Operation	**Return type**	**Description**
generateUpdateOne (String parameters, ComputationalModel model)	ArrayList<String>	Generates the subroutine update_one. Note: This method has not been implemented and it does nothing!
generateUpdateAll (String parameters, ComputationalModel model) throws InvalidModelException, IllegalParametersException, MissingFunctionException	ArrayList<String>	Generates the subroutine update_all.
setSpatialStartValue (Variable var) throws InvalidModelException	ArrayList<String>	Generates code that calculates the initial value for a spatial variable var.
setFunctionalStartValue (Variable var)	ArrayList<String>	Generates code that calculates the initial value for a functional non-spatial variable var.

5.4.2 Acceptation

Acceptation generates parts of the subroutine update_all: new value generation, acceptation formulae and acceptation codes (things to do when a proposal is accepted). FortranMain calls the methods of this class.

Public methods

Operation	**Return type**	**Description**
setTopologicalList (ArrayList<Variable> list)	void	Sets the topological variable list needed in the following methods. This method should be called before calling any other methods of this class.
generateNewValueCode (Variable variable)	ArrayList<String>	Generates code that fetches a proposed new value for a parameter from its buffer or if the buffer is empty, calls the Fortran subroutine 'generate'.
generateNewValues Functional-Code(Variable variable) throws InvalidModelException	ArrayList<String>	Generates code that calculates new values for functional parameters depending on the variable.
generateAcceptationFormula (Variable variable) throws InvalidModelException	ArrayList<String>	Generates code that calculates the acceptation probability for the new value.
generateAcceptationCode (Variable variable, String strategy) throws InvalidModelExcpetion	ArrayList<String>	Generates code that decides whether the new value is accepted and makes the necessary changes: updates the value of the current parameter and all functional parameters depending on it.

Private methods

Operation	**Return type**	**Description**
generateSpatialDeps (Variable functional, String indexing, boolean spatialY)	ArrayList<String>	Sets correct indexing for functional variables with spatial dependencies. For example: q = sum(&x) and p = q/2.0. When x is updated (the spatial target) first q is calculated for all neighbours of x, then all neighbours of x get a new p to match their new q values. This method is used when generating the new value calculation for p. This methods doesn't generate any loops.

Operation	Return type	Description
acceptationAffectedOf (Variable variable, Variable depending) generateUpdateOneFunctional (Variable depending)	ArrayList<String> ArrayList<String>	Generates code which updates the value of depending when the value of variable is accepted. Depending is assumed to be non-spatial. Generates code that replaces the value of depending with its new value.
generateNewValueForOne Functional (Variable functional, Set<Variable> newToBeUsed) throws InvalidModelException	ArrayList<String>	Generates code which calculates the new value for functional by using the new values of the variables belonging to the set newToBeUsed and the current values of other variables.
generateAcceptationFormula Global (Variable variable)	ArrayList<String>	Generates code that calculates the acceptation probability for a global variable.
generateAcceptationFormula OneDimensional (Variable variable)	ArrayList<String>	Generates code that calculates the acceptation probability for an one-dimensional variable.
generateAcceptationFormula TwoDimensional (Variable variable)	ArrayList<String>	Generates code that calculates the acceptation probability for a two-dimensional variable.
generateLikelihoodFormula Global (Variable depending, Set<Variable> newToBeUsed)	ArrayList<String>	Generates code that calculates the likelihood probability of a global variable depending by using new values of the variables belonging to newToBeUsed, that is: P(depending | new values of variables in newToBeUsed) / P(depending | values of variables in newToBeUsed).
generateLikelihoodFormula OneDimensional (Variable depending, Set<Variable> newToBeUsed)	ArrayList<String>	Generates code that calculates the likelihood probability of an one-dimensional variable depending by using new values of the variables belonging to newToBeUsed, that is: P(depending | new values of variables in newToBeUsed) / P(depending | values of variables in newToBeUsed).
generateLikelihoodFormula TwoDimensional (Variable depending, Set<Variable> newToBeUsed)	ArrayList<String>	Generates code that calculates the likelihood probability of a two-dimensional variable depending by using new values of the variables belonging to newToBeUsed, that is: P(depending | new values of variables in newToBeUsed) / P(depending | values of variables in newToBeUsed).
generateTransitionFormula (Variable variable)	ArrayList<String>	Generates code that calculates the transition probability for variable, that is q(variable', variable) / q(variable, variable').

5.4.3 Input

This class is used to generate the Fortran module 'input'. See 4.2.2.

Public methods

Operation	**Return type**	**Description**
generateInput (String callParameters, ComputationalModel model) throws IOException, SyntaxException	void	Generates and writes the Fortran module input.f90.

Private methods

Operation	**Return type**	**Description**
generateReadData (LinkedList<Variable> variables, LinkedList<Entity> entities, String callParameters) throws SyntaxException	ArrayList<String>	Generates the subroutine read_data.
generateSetInitialValues (String file, LinkedList<Variable> variables, LinkedList<Entity> entities, String callParameters)	ArrayList<String>	Generates the subroutine setInitialValues.
generateReadSpatial (int neighbours, LinkedList<Entity> entities	ArrayList<String>	Generates the subroutine set_spatial.

5.4.4 Output

This class is used to generate the Fortran module 'output'. See 4.2.3.

Public methods

Operation	**Return type**	**Description**
generateOutput (String callParameters, ComputationalModel model) throws IOException	void	Generates and writes the Fortran module output.f90.

Private methods

Operation	Return type	Description
generateWriteOutput (ComputationalModel model, String callParameters)	ArrayList<String>	Generates the subroutine write_output.
generateWriteSummary (ComputationalModel model, String callParameters)	ArrayList<String>	Generates the subroutine write_summary.
generateWriteLastValues (ComputationalModel model, String callParameters)	ArrayList<String>	Generates the subroutine write_last_values.

5.4.5 Proposal

This class is used to generate the Fortran module ‘proposal’. See 4.2.

Operation	Return type	Description
generateProposal (ComputationalModel model) throws InvalidProposalException, IllegalParametersException, MissingFunctionException, IOException	void	Generates and writes the Fortran module proposal.f90.

6 Correspondence to requirements

This chapter describes the correspondence between requirements found in SRS document and the implementation of the software.

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

Possible implementation statuses include:

- Implemented: The software supports the requirement
- Designed but not implemented: The requirement was designed to be implemented but it was dropped at the implementation phase
- Not implemented: The was not designed and it was not implemented

6.1 Model requirements

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

6.2 Data requirements

Requirement	**Priority**	**Implementation status**
D1: The general data format	E	Implemented
D2: Data not available	E	Implemented
D3: Invalid data	E	Implemented

6.3 Simulation requirements

Requirement	**Priority**	**Implementation status**
S1: The algorithm used	E	Implemented
S2: Choice of algorithm	C	Not implemented
S3: Setting the number of updates	E	Implemented (except when related to blocks)
S4: Setting the number of burn-in iterations	E	Implemented
S5: Setting the thinning factor	E	Implemented
S6: Setting the blocks	C	Not implemented
S7: Setting the update strategy	C	Designed but not implemented
S8: Setting the weight of the blocks	O	Not implemented
S9: Setting the proposal strategies for variables	E	Implemented
S10: Proposal distributions	E	Implemented
S11: Setting initial values	E	Implemented
S12: Defining parameters to output	E	Implemented
S14: Informing the user about the progress	E	Implemented
S15: Soft stop	O	Not implemented
S16: Parameters in random walk	O	Not implemented

6.4 Output requirements

Requirement	**Priority**	**Implementation status**
OP1: Writing output into a file	E	Implemented
OP2: Output file names	E	Implemented
OP3: The output	E	Implemented
OP4: Information written to output files	E	Implemented
OP5: Summary of the simulation	C	Implemented
OP6: File access check	C	Implemented

6.5 General error conditions

Requirement	**Priority**	**Implementation status**
E1: File not found	E	Implemented
E2: Reporting syntax errors	O	Implemented
E3: Reporting semantic errors	O	Not implemented

6.6 Non-functional requirements

Requirement	Priority	Implementation status
N1: Working on Linux	E	Implemented
N2: The implementation language	E	Implemented
N3: Parallel computation	O	Not implemented
N4: Graphical user interface	O	Not implemented

6.7 General requirements

Requirement	Priority	Design status
G1: Adding comments to definition files	E	Implemented

7 Future development

The following subsections describe features that could be added to the software and define which parts of the program they would affect. The features which could be implemented easily are described at first.

7.1 Random update strategy

Requirement S7

The only change necessary would be to add the method generateUpdateOne to FortranMain. Other classes and methods would remain unchanged. (For example the classes Definitions, Acceptation and the Parser already support the random update strategy.)

The prototypes have the subroutine update_one implemented so that they could be used when implementing the method generateUpdateOne.

This feature was designed but not implemented.

7.2 More distributions

Requirement M10

More distributions could be added by modifying the Fortran module user_dist.f90. No changes to the Generator would be needed. Some subroutines are trivial to implement by using existing NAG subroutines and functions while other subroutines are more difficult.

7.3 Making the generating quicker

If some input files (for example data) are not changed since the previous run the generator wouldn't have to check their validity. There are multiple ways to implement this feature.

7.4 Defining properties for single parameters

The format of the input files should be changed so that it would be possible to define a proposal distribution or an update count for a single parameter instead of parameter groups. The Parser should be changed accordingly. The data structures and Generator classes already support these features.

7.5 The Gibbs algorithm

Requirement S2

When using the Gibbs algorithm for a parameter its proposal distribution may contain references to other variables. The proposed value is accepted without calculating its likelihood.

Adding the Gibbs strategy would change the structure of the Fortran program. It would be impossible to generate new value proposals in advance, since the proposal distributions would change during the simulation.

The module “proposal” would no longer be able to generate proposals for all the parameters. The parameters with a static proposal strategy could utilize it as before. New values for other parameters should be generated in the subroutine update_all (or update_one if the random update strategy is used), since it would be too inefficient to add perform a subroutine call every time a proposal is needed. The acceptation procedures would also change a little.

The data structures Distribution and Variable are flexible enough to allow the description of non-static proposal distributions.

The Parser would have to be changed to allow variable references when defining proposal distributions.

The classes Acceptation and FortranMain would have to be changed to generate code which updates also the parameters with the Gibbs algorithm.

The class Variable already has a field for algorithm choice.

7.6 Parameters of proposal distribution

Requirement S16

The parameters of proposal distribution could depend on the parameter’s current value when using the random walk proposal strategy.

The implementation should be straightforward if the Gibbs algorithm is supported since the parameter can be a parameter of its own proposal distribution.

7.7 Parameter blocks

Requirements S6, S8

This feature was not designed. It would be necessary to introduce a data structure describing a block and changes to almost every class would be needed.

7.8 Soft stop

Requirement S15

By pressing a defined soft-stop key the user could be able to interrupt the simulation so that the output files would be at consistent state (for example with only fully executed iterations and the last values file existing).

The soft-stop feature could be added by editing the method generateMain in class FortranMain. It hasn't been determined if the soft stop is possible to be implemented in Fortran (that is, if it is possible to determine if a certain key is pressed).

7.9 Graphical user interface

Requirement N4

A graphical user interface for drawing the models and defining other parameters could be added to the generator.

7.10 Reporting semantic errors

Requirement E3

The Generator could report semantic errors in model descriptions. It is not straightforward to determine whether a model is semantically invalid.

7.11 Parallel computing

Requirement N3

The Fortran program could be designed to utilize multiple processors. For example generating proposals could be done in parallel with simulation. It has not been determined which other parts of the execution could be done in parallel.

8 References

NAG	NAG Fortran library (Mark 19) http://www.csc.fi/cschelp/sovellukset/math/nag/NAGdoc/fs/html/genint/libconts_fs20.html