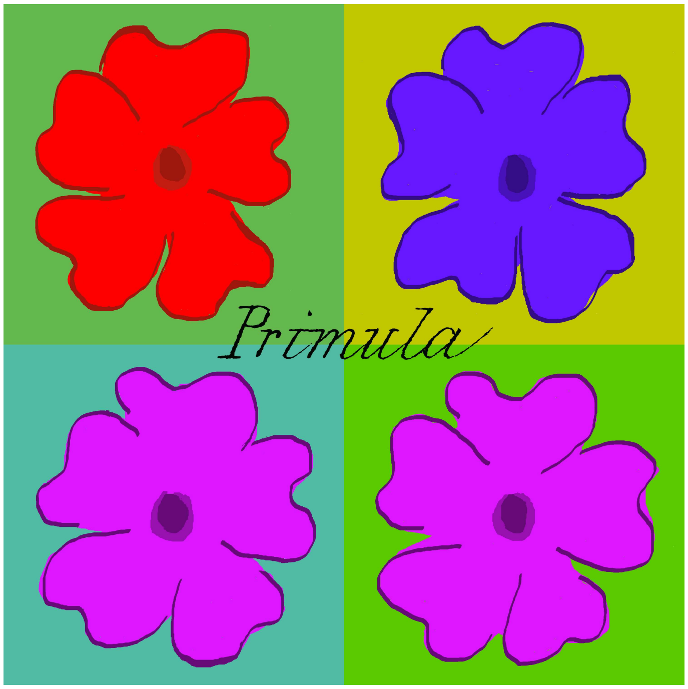
# The *Primula* System: user's guide
# Version 2.2

Manfred Jaeger
Institut for Datalogi, Aalborg Universitet, Selma Lagerløfs Vej 300, 9220 Aalborg Ø
jaeger@cs.aau.dk

*Primula* homepage: www.cs.aau.dk/∼jaeger/Primula

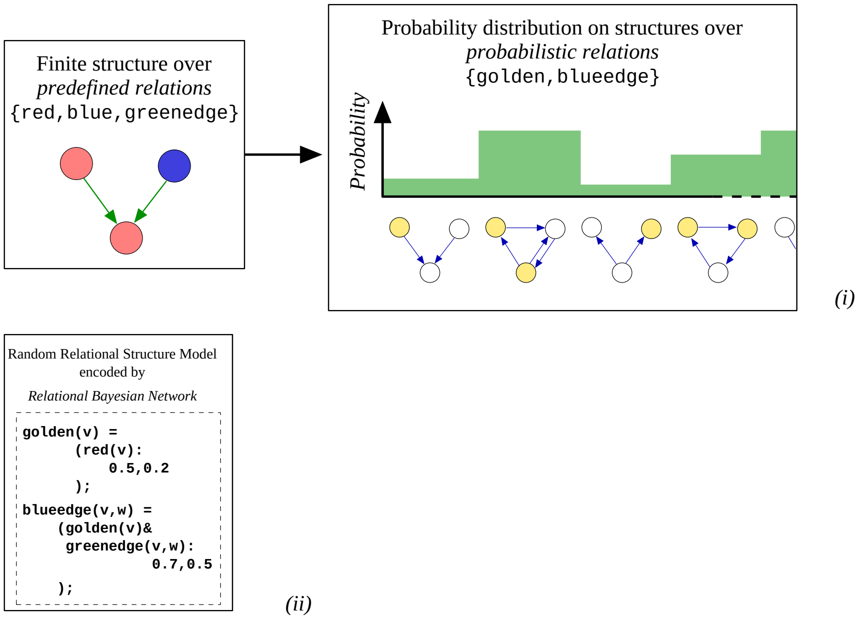
June 18, 2009



# Contents

# 1 Introduction

*Primula* is a Java-based software system for modelling and inference with *random relational structure models*. Random relational structure models are a natural class of probabilistic models that subsumes classical models like Markov chains and random graphs, as well as the more recent developmentsof dynamic Bayesian networks and probabilistic relational models [7].


Finite structure over
*predefined relations*
{red,blue,greenedge}
Probability distribution on structures over
*probabilistic relations*
{golden,blueedge}
*Probability*
*(i)*
Random Relational Structure Model
encoded by
*Relational Bayesian Network*
*(ii)*

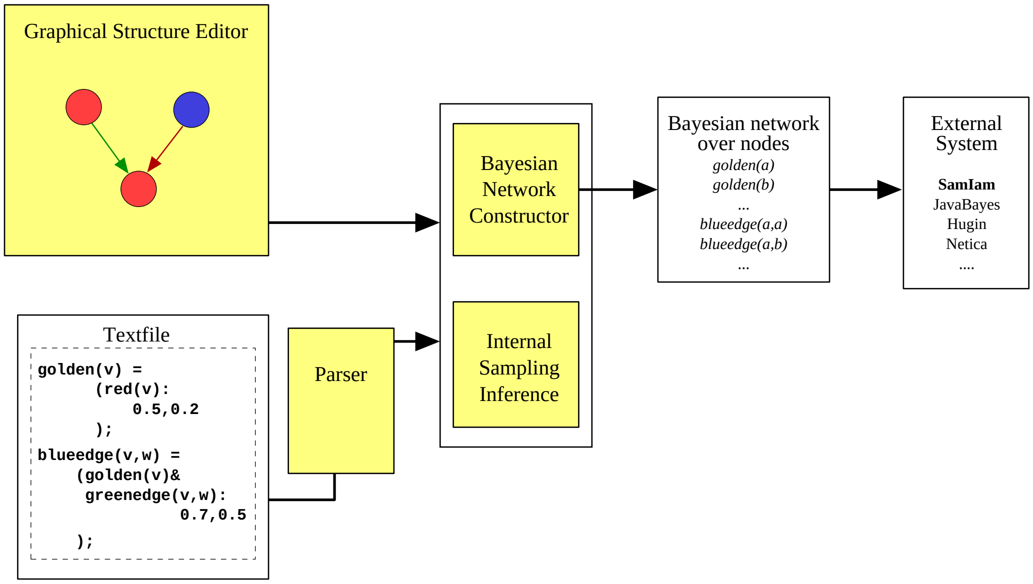
```
golden(v) =
      (red(v):
          0.5,0.2
      );
blueedge(v,w) =
    (golden(v)&
     greenedge(v,w):
               0.7,0.5
    );
```

Figure 1: Random Relational Structure Model

Figure 1 *(i)* gives a schematic description of a random relational structure model: it is a mapping that takes finite relational structures over some *predefined relations* as arguments, and returns probability distributions over all relational structures over a different set of *probabilistic relations*. A practical example is a general (probabilistic) model of inheritance (like Mendelian laws): such a model induces a mapping that takes known relationships (encoded in the relational structure of a family tree) as inputs, and returns a probability distribution on the presence or absence of certain genetic traits (formalized as unary probabilistic relations) in the members of the family.

A formal representation language for random relational structure models are *relational Bayesian networks*, introduced in [10]. *Primula* is an implementation of this language that uses the inference-techniques described in [11]. Figure 2 gives an overview of the system, and its main components: a graphical structure editor is used to specify the relational input structure. A relational Bayesian network is read from a file and transformed into the appropriate internal data structures by a Parser. Three different methods for probabilistic inference are then available:

- A standard Bayesian network can be constructed that represents the distribution of the probabilistic relations for the given particular input structure. This Bayesian network can be
  - directly exported to the SamIam system (http://reasoning.cs.ucla.edu/samiam/). SamIam version 2.3 beta or higher is required.
  - exported as a file in various formats, which then can be read by a number of other Bayesian network systems.
- the built-in sampling algorithms can be used for approximate inference.
- exact inference via the built-in interface to the ACE system (http://reasoning.cs.ucla.edu/ace/). ACE must be separately downloaded and installed.



```
golden(v) =
      (red(v):
          0.5,0.2
      );
blueedge(v,w) =
    (golden(v)&
     greenedge(v,w):
               0.7,0.5
    );
```

Figure 2: Components of the *Primula* System

## 2 Some Introductory Examples

In this section we first give an informal overview over *Primula*'s modelling capabilities by several examples. All input files for these examples are distributed with the *Primula* system.

## Mendelian Model of Inheritance

Suppose that some gene has the two alleles $A$ and $a$. One will then distinguish genotypes $A/A$, $A/a$, and $a/a$. In principle, one may also make the four-way distinction $A/A$, $A/a$, $a/A$, and $a/a$, where the first letter represents the allele inherited from the father, the second letter the allele inherited from the mother. According to Mendel's laws, each parent will pass on to its offspring with equal probability either of its alleles, independently for different offsprings, and independently from the allele passed on by the other parent. Mendel's laws provide a general probabilistic model for the distribution of genotypes in a population. This general probabilistic model can be instantiated to a concrete probabilistic model when a concrete population with its family relationships (i.e. a family tree, or pedigree) is given.

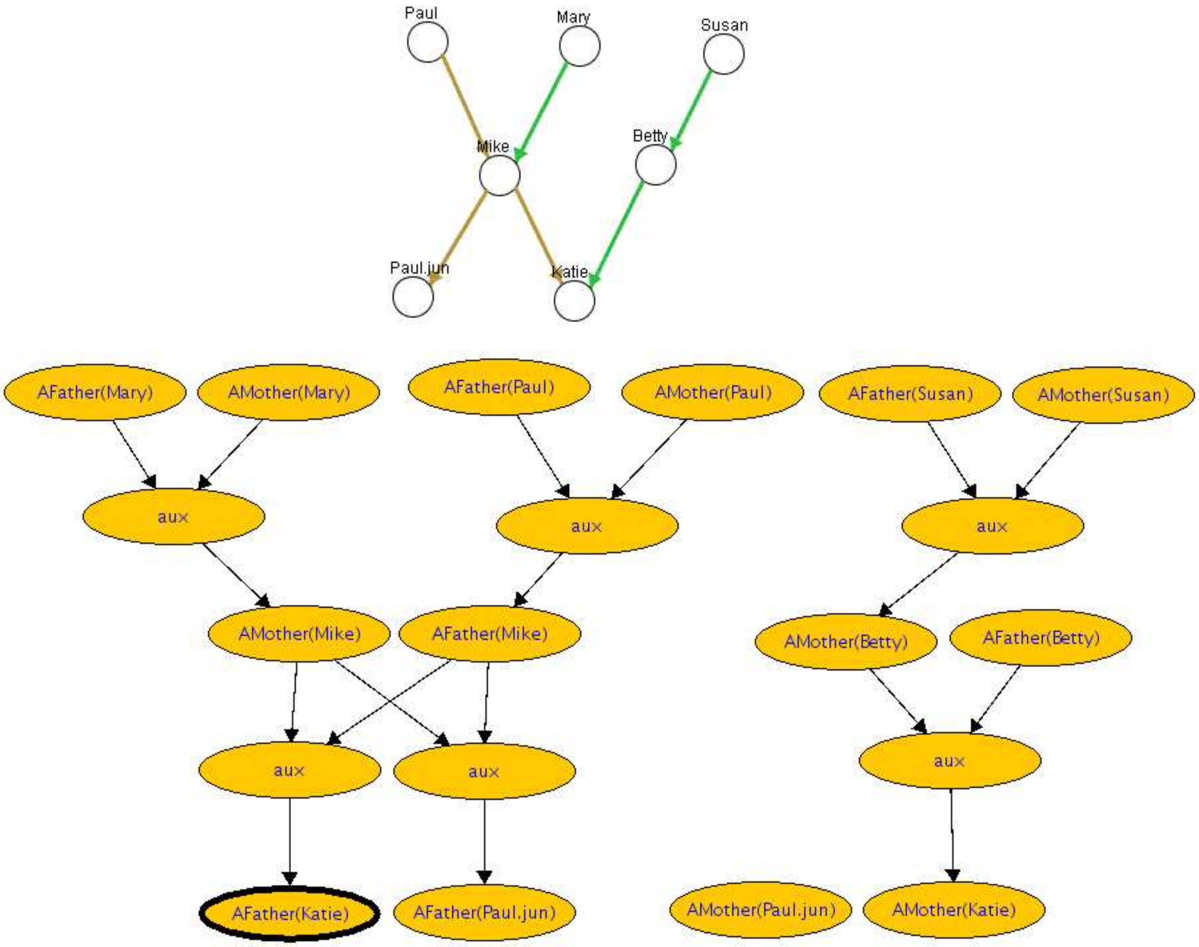


Figure 3: Pedigree and Genotype Model

Figure 3 shows a family tree of seven individuals, and a probabilistic model for the genotypes in this family represented by a Bayesian network (all illustrations of Bayesian networks in this document are screenshots from the SamIam system). This is a model for the ordered genotype, where the fact that $A$ was inherited from the father is represented by the propositional variable `AFather`. The (ordered) genotype a/A, thus, corresponds to the truth assignments `AFather`=false, `AMother`=true. Mendel's laws only describe how genetic traits are inherited. In order to specify a probabilistic model one also has to assign probabilities for the genotypes of individuals for whom not both parents are known. This can be done, for instance, by assigning a constant ”base-rate”probability to AFather or AMother being true. Table 1 shows an encoding of the resulting model in the Relational Bayesian Network language (here assuming a base-rate probability 0.3 for allele $A$).

Bayesian networks generated by *Primula* typically contain a number of auxiliary nodes in addition to the nodes representing the probabilistic relations. These auxiliary nodes are inserted to decompose large conditional probability tables into several smaller ones (see [11] for the details). Depending on the Bayesian network system used for processing the generated standard Bayesian network, these auxiliary nodes will either be labelled ’aux’, or with their automatically generated internal identifiers.

The language of Relational Bayesian Networks will be introduced in detail in section 8. For now it is sufficient to understand that this is a formal representation language, which, depending on one’s preferences, can be seen either as a functional programming language, or as a probabilistic variant of predicate logic formulas. With the *Primula* system one can

- specify general probabilistic models in the language of Relational Bayesian Networks
- specify relational structures that instantiate the general model
- automatically construct a Bayesian network that represents the instantiated model, and that can be used for probabilistic inference in that model.

```
@fatherInTree(v) = n-or{sformula(father(u,v))|u:  u=u};
@motherInTree(v) = n-or{sformula(mother(u,v))|u:  u=u};

AFather(v) = ( @fatherInTree(v):
                      mean{AFather(u),AMother(u)|u:  father(u,v)},
                      0.3
               );

AMother(v) = (@motherInTree(v):
                      mean{AFather(u),AMother(u)|u:  mother(u,v)},
                      0.3
               );
```

Table 1: Relational Bayesian Network Representation of Mendelian Model

## Linkage Analysis

More elaborate genetic inheritance models are used in *linkage analysis* for locating in the (human) genome the position of genes coding for certain observable traits. SUPERLINK [6] is a Bayesian network based system for performing probabilistic analyses in linkage analysis (http://bioinfo.cs.technion.ac.il/superlink).

SUPERLINK requires two inputs: a *locus file* containing the specification of the type of genetic information available for the current analysis, and a *pedigree file* containing pedigree information on a number of individuals for whom this genetic information was obtained. Both inputs together determine a probability distribution on unobserved genetic variables. The SUPERLINK system in conjunction with any fixed locus file determines a random relational structure model that maps pedigrees to a probability distribution over the genetic variables for each individual in the pedigree. An example for such a random relational structure model is contained in `linkage_3loci.rbn`.

The model in `linkage_3loci.rbn` operates on essentially the same relational input structures as `mendel.rbn`. However, to simplify the model code, it is here assumed that for all individuals in the pedigree either no or both parents also are in the pedigree, and that in the former case the individual is marked with the unary *founder* relation. Some example input domains for the model are contained in `linkagefamily_x.rdef` files (x=s,m,l).

## Dynamic Bayesian Network

A dynamic Bayesian network specifies a general temporal probabilistic model by means of network fragments that can be concatenated to form probabilistic models for arbitrary discrete, finite time sequences. A time sequence of $n$ (discrete) time points can be represented as a relational structure with a binary *predecessor* relation. The temporal model represented by network fragments can also be encoded as a Relational Bayesian Network. A very simple example with two boolean variables `a` and `b` at each timepoint is contained in `dynamic.rbn`.

## Probabilistic Relational Model

*Probabilistic Relational Models* in the sense of Friedman et al. [7] can also be handled in the *Primula* system. As an example, we here show how to encode a model described in [8]. This model distinguishes three classes of objects: Movies, Theaters, and Shows. Objects of these classes have various (probabilistic) attributes: Movies have attributes Genre, Budget, Decade (of origin), Theaters have attributes Type and Location, and Shows have attributes Theater and Movie. The core modelling problem here is how to define a probability distribution over what movies are shown by which theaters. This model should be generic in the sense that it can be instantiated by arbitrary sets of objects from the different classes.

In our representation of this model, we do not consider Shows as a class of objects with attributes Movies and Theaters, but more directly as a binary relation between Movies and Theaters. A concrete instantiation of the general model, then, is given by sets of objects from the Movies and Theaters classes only. An example is shown in figure 4. We here specify both class membership and attribute values like Genre = *thriller* by unary relations, which, in turn, in the graphical representation are color-coded (for instance, objects in class Theater are colored dark green, objects with Genre=*thriller* are colored blue, Genre=*foreign* is represented by yellow, etc.). Concentrating on the probabilistic binary relation `shows`, we here take all the attributes as predefined, even though they, too, could be modelled probabilistically.

The Relational Bayesian Network contained in `movies.rbn` now specifies the probabilistic model described in [8]: a theater chooses the movie it shows either uniformly from the set of movies with Genre =

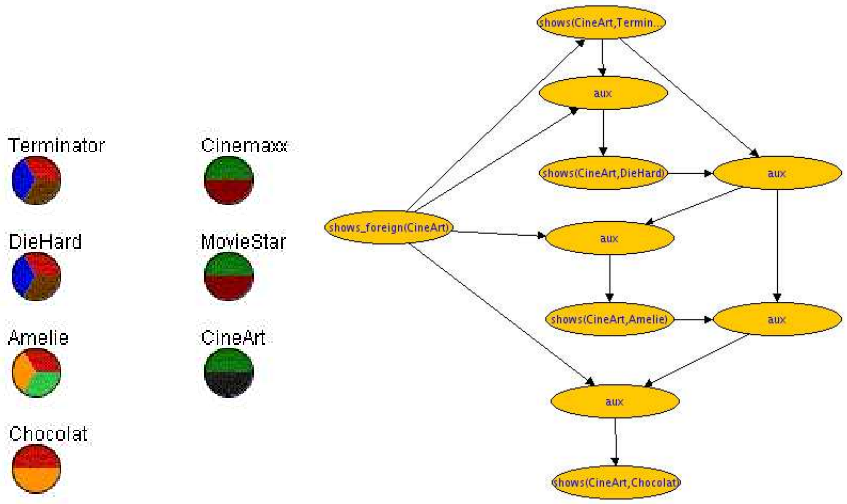


Figure 4: Movies and Theaters

*thriller*, or uniformly from the set of movies with Genre=*foreign*. A theater of type *megaplex* will choose with probability 0.1 from the *foreign* movies, and with probability 0.9 from the *thriller* genre. For an *art theater* these probabilities are 0.7 and 0.3, respectively. In our example, theater MovieStar is of type megaplex. Figure 4 now shows a Bayesian network for the probability distribution on which movie is shown at CineArt.

The complete output of *Primula* when run on `movies.rbn` and the input structure shown in figure 4 consists of three separate networks, one for each of the three cinemas.

## Robot Navigation Model

Figure 5 shows a simple grid-map that distinguishes five different *locations*, and specifies their spatial relationships with two binary relations *leftof* and *belowof*. Also shown in the figure is an object of type *Block*. The Relational Bayesian Network `randblock.rbn` describes the following model: each block is positioned randomly at one location on the map (the general model can also be instantiated with more than one *Block* object). This positioning is represented by the binary relation `blocks` between blocks and locations. A binary relation `connectionblocked` between locations is deterministically defined in terms of `blocks`: `connectionblocked`$(l_1, l_2)$ is true if at least one of the two locations $l_1, l_2$ is blocked.The intuition here is that a robot can move between $l_1$ and $l_2$ iff `connectionblocked`$(l_1, l_2)$ is false. Also shown in Figure 5 is the Bayesian network constructed for this instance.

A more ambitious version of this model contained in `randblock_trans.rbn` extends the model with a (again deterministic) definition of the transitive closure `connected` of `connectionblocked`. As with this model we push even beyond the limits of first-order logic (transitive closure is not first-order definable), it is not surprising that inference in the resulting model becomes quite costly, and currently does not appear feasible for all but very small maps.

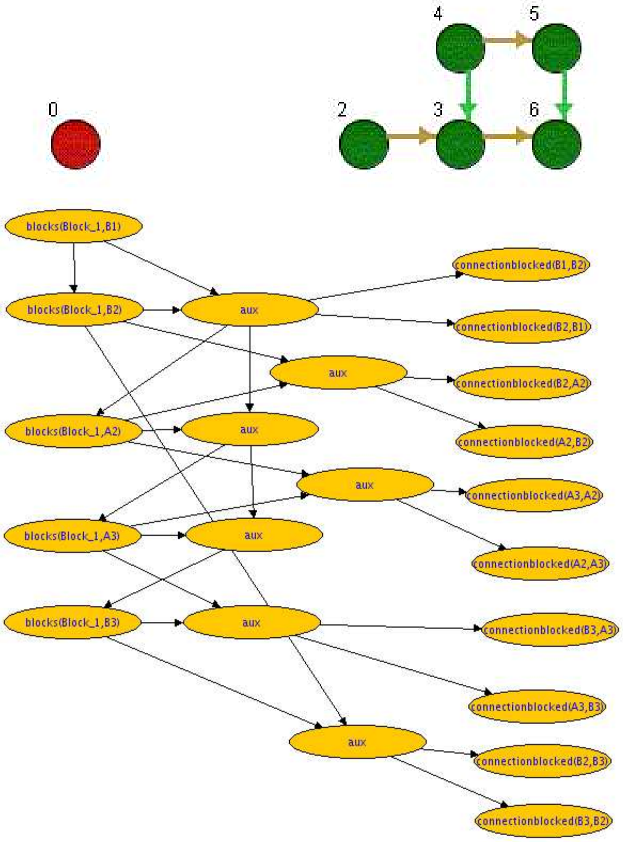
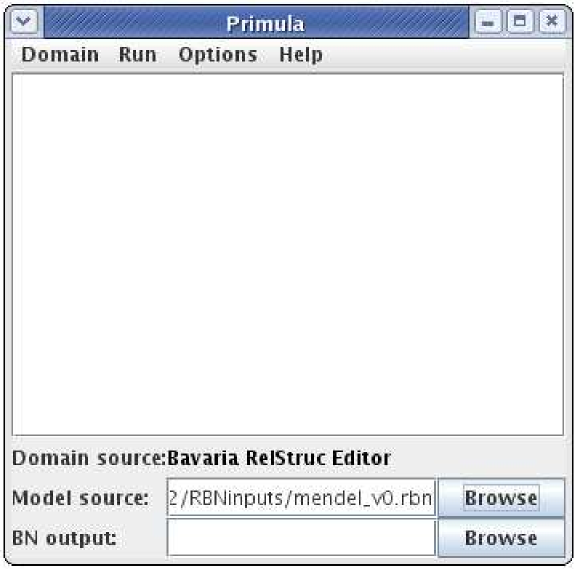


Figure 5: Grid Map and Random Block

# 3 The *Primula* Console

Figure 6 shows the console of the *Primula* system. Only this window will be opened when the system is started. Main functions of the Primula console are the choice of input sources for relational structures and Relational Bayesian Networks, and the setting of options.



Domain source:Bavaria RelStruc Editor

Model source: 2/RBNinputs/mendel_v0.rbn

BN output:

Figure 6: *Primula* Console

At the bottom of the window are two file choosers: one for the textfile containing the Relational Bayesian Network, and one for the output file for the constructed (standard) Bayesian network. The relational input domain can be specified in several ways, not only by reading from a file. Therefore the *Primula* console displays at the bottom only a line describing the current input mode for the input domain, while the actual choice is via the Domain menu.

## Domain menu

**Create OrderedStruc** With this option one can define as the input structure an ordered structure of a specified size. An ordered structure of size $n$ n has elements $0, ..., n-1$ on which the binary relations *less* (i.e. $<$ according to the natural ordering), *pred* (i.e. *pred*$(i,j)$ iff $i+1=j$), and the unary relations *zero* (*zero*$(i)$ iff $i=0$) and *last* (*last*$(i)$ iff $i=n-1$) are defined.

**Load Relational Structure** Load a relational structure in .rdef or .rst format from file.

**Start Bavaria** Start the Bavaria structure editor, and use current structure in the editor as input structure.

## Run menu

**Start Inference Module** Open the Inference Module (see below).

**Start Learn Module** Open the Learning Module (see Section 6).

**Construct Bayesian Network** Construct the standard Bayesian network according to the current input structure and current Relational Bayesian Network specification. Write result to file specified in BN output field or open in SamIam.

**Save RBN** Saves the current RBN model into .rbn file. Mostly used when current model has learned parameter values.

**Convert Relational Data** Starts an interactive dialog to transform data in Prolog format into rdef. See Appendix B.1.

**Open SamIam** Start the SamIam Bayesian network system (if installed).

*Notes on using SamIam:*

- To use Primula with the direct SamIam interface, SamIam version 2.3-beta or higher must be installed. SamIam is available free at http://reasoning.cs.ucla.edu/samiam/
- When first calling SamIam from Primula, a file selection dialog will be opened in which the locations of the `inflib.jar` *and* `samiam.jar` archives of the SamIam system have to be specified.
- `.rbn` input files should contain the underscore _ as the only special character. This is because *Primula*and SamIam communicate using some of HUGIN's file format and naming conventions, and HUGIN names follow the lexical rules of C identifiers.

## Options menu

### Bayesian Network System

Choose the export mode for the generated Bayesian network. If **To SamIam** is chosen, the SamIam system will be automatically started, and the generated network opened in SamIam. If **Java Bayes, Hugin** or **Netica** is chosen, the network will be exported as a file in .bif format (JavaBayes: http://www-2.cs.cmu.edu/∼javabayes/Home/), .net format (Hugin: http://www.hugin.com/), respectively .dne format (Netica: http://www.norsys.com/). In this case, an output file must be specified in the **BN Output** field in the console (not required under the **To SamIam** option).

Since the different file formats offer different capabilities, there are some slight differences in *Primula*'s output, depending on the chosen Bayesian Network System:

- The .bif format does not support node labellings that are different from the unique internal node identifiers. Auxiliary nodes that are generated in the network construction, but do not represent random variables that are part of the intended probabilistic model, are displayed labelled ”aux” when .net or .dne is chosen, and labelled with rather lengthy strings when .bif is chosen.
- The .net format does not support instantiation information of nodes. See below for the consequences when the ”Evidence Conditioned” option is chosen.

**Construction Mode**

**Query Specific** Only construct a Bayesian network that is sufficient to compute the probabilities of query-atoms specified with the inference module.

**Evidence Conditioned** Integrate the instantiation of selected atoms as specified in the inference module into the network construction.

*Note:* The resulting network only encodes the correct probabilities as long as the instantiations used in the construction are unchanged (but further evidence may be added in the constructed network).

*Note:* As the .net format does not allow to pass the information of instantiations used in the constructionon to the Bayesian network system (Hugin), it is only possible to use the 'Hugin' option together with the 'Evidence Conditioned' option when all instantiations used in the network construction are also manually entered into the Bayesian network.

**Skip layout:** In the construction of the Bayesian network no node positions for a graphical display of the network are computed. This can substantially reduce computation time. When displayed in a Bayesian network system, all nodes will appear stacked on each other in one place.

**Show isolated prob.0 nodes** A generated Bayesian network may contain nodes representing ground atoms that are deterministically false and not connected to any other nodes. With this option it can be chosen whether to include such nodes in the generated model or not.

**Decompose mode**
**normal**: The standard decomposition technique described in [11] is used. The in-degree of all nodes in the generated Bayesian network is bounded by 3. This decomposition option is only available for models that only contain *multi-linear* combination functions (cf. Table 7 on p. 30).

**deterministic**: An extended decomposition is applied that results in a network where all internal (non-root) nodes have deterministic cpts. The in-degree is bounded by 4. This option also requires multi-linear combination functions, and will be of little use to most users.

**none**: No decomposition is applied. The in-degree of nodes is not bounded and can grow polynomially in the size of the input structure. Main advantage of this option is that it permits the use of combination functions that are not multilinear.

**Reset external software locations**
Resets (to “unknown”) previously entered locations of external software components (SamIam and ACE). Useful when new versions of external software have been installed.

# 4 The Inference Module

The inference module is used:

- For exact inference: to enter evidence and queries, which can greatly simplify the Bayesian networks that will be constructed (options “evidence conditioned” and/or “query specific” must be selected in the options menu of the *Primula* console).
- For exact inference using ACE.
- For approximate inference: as an interface to the sampling methods.

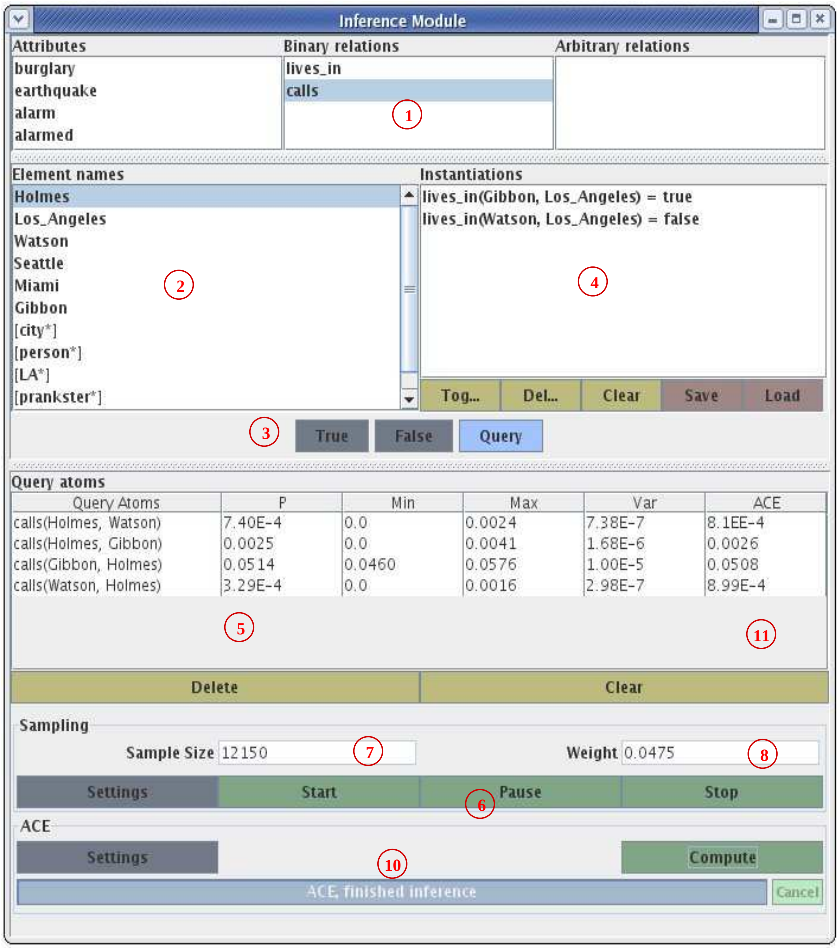
## Entering Evidence and Queries

The evidenc module displays in a similar form as the Bavaria module all probabilistic relations defined by the currently loaded Relational Bayesian Network $\boxed{1}$, and all object names defined by the current relational input structures $\boxed{2}$. By choosing one of the buttons ’True’, ’False’, ’Query’, $\boxed{3}$, atoms can be instantiated to a truth value, or specified as query atoms. Entered instantiations can be seen and edited in the ’Instantiations’ panel $\boxed{4}$. Queries are shown in the ’Query atoms’ panel $\boxed{5}$.

To enter evidence `calls`(*Watson,Holmes*)=true, select the ’True’ button, then select `calls` in the list of relations, and then click on Watson and Holmes in the list of Element names.

The “Element names” list also contains wildcards for all objects belonging to a given unary predefined relation. When the ’False’ button and the `calls` relation are selected, for example, then clicking first on [person*] and then on Holmes in the Element names list, will add all instantiations `calls`($p$,*Holmes*)=*false* to the Instantiations list, where $p$ ranges over all persons in the input domain.

To illustrate the use of the inference module, and the ’Query Specific’ and ’Evidence Conditioned’ options, Figure 8 shows the network constructed when both options are chosen, and evidence and query specifications are as in Figure 7. Note how instantiating `lives-in`(*Watson,Los-Angeles*) to false eliminates the dependency of `calls`(*Holmes,Watson*) on `earthquake`(*Los-Angeles*), whereas instantiating `lives-in`(*Gibbon,Los-Angeles*) to true has no such effect.



Attributes: burglary, earthquake, alarm, alarmed

Binary relations: lives_in, calls

Element names: Holmes, Los_Angeles, Watson, Seattle, Miami, Gibbon, [city*], [person*], [LA*], [prankster*]

Instantiations:

lives_in(Gibbon, Los_Angeles) = true
lives_in(Watson, Los_Angeles) = false

| Query Atoms | P | Min | Max | Var | ACE |
|---|---|---|---|---|---|
| calls(Holmes, Watson) | 7.40E-4 | 0.0 | 0.0024 | 7.38E-7 | 8.1EE-4 |
| calls(Holmes, Gibbon) | 0.0025 | 0.0 | 0.0041 | 1.68E-6 | 0.0026 |
| calls(Gibbon, Holmes) | 0.0514 | 0.0460 | 0.0576 | 1.00E-5 | 0.0508 |
| calls(Watson, Holmes) | 3.29E-4 | 0.0 | 0.0016 | 2.98E-7 | 8.99E-4 |

Sample Size 12150

Weight 0.0475

Figure 7: Inference Module

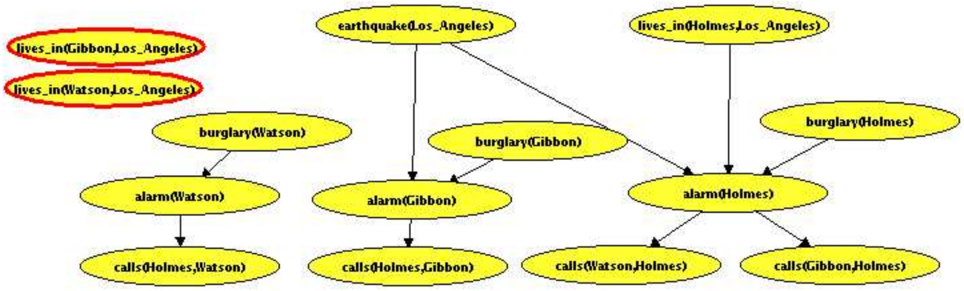


Figure 8: Evidence conditioned Holmes network

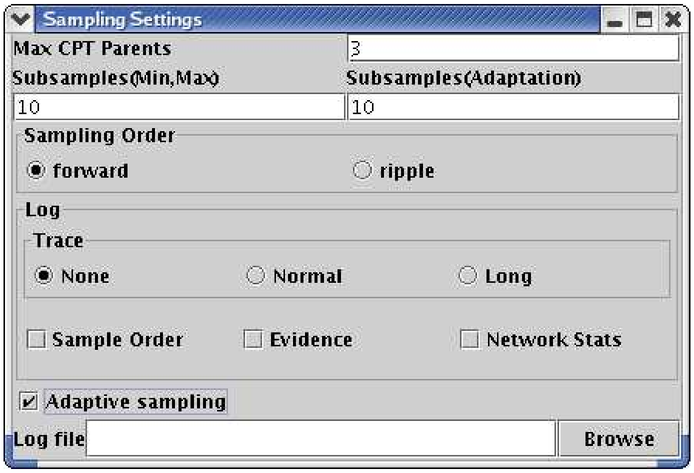
## Sampling

The inference module also provides the interface to *Primula*'s own sampling based inference, which is based on (adaptive) importance sampling. Sampling inference only computes approximate probability values for the specified query atoms, so at least one atom has to be defined as a query atom. The option “Query specific” from the Options menu of the Primula console has no effect on sampling inference. The option “Evidence conditioned” in Primula's Options menu, on the other hand, determines whether the probabilities computed by the sampling procedure are conditional on the evidence entered in the inference module.

Sampling is started by clicking on the ’Start’ button in the sampling panel $\boxed{6}$. *Primula* will initialize a sampling process and continue sampling until the ’Pause’ or ’Stop’ button is pressed. At regular update intervals (approximately every 2 seconds) the probability estimates based on the current sample will be updated in the ’Query atoms’ tabel $\boxed{5}$. Furthermore, the size of the current sample is displayed $\boxed{7}$. The ’Weight’ displayed in $\boxed{8}$ is an estimate of the marginal probability of the entered evidence (if option ’Evidence conditioned’ is chosen, otherwise it is constant 1.0).

The ’P’ column in the Query atoms table shows the current point estimate for the probability of the query atom. ’Min’ and ’Max’ show the minimal, respectively maximal estimate obtained by dividing the current sample into a number of subsamples (this number is specified in ’Sampling Settings’). ’Var’ shows the empirical variance of the estimates obtained from the subsamples. If, for example, ’Subsamples’ is set to 10, and the current sample size displayed in $\boxed{7}$ is 10000, then ’Min’, ’Max’ and ’Var’ refer to the 10 samples of size 1000 each (each sample is directly assigned in the sampling process in a round-robin fashion to one of the 10 subsamples).

### Sampling Settings



Sampling Settings

Max CPT Parents: 3

Subsamples(Min,Max): 10

Subsamples(Adaptation): 10

Sampling Order: ● forward ○ ripple

Log

Trace: ● None ○ Normal ○ Long

□ Sample Order □ Evidence □ Network Stats

☑ Adaptive sampling

Log file

Figure 9: Sampling Settings

Pushing the “Settings” button in the Sampling panel of the Inference Module opens the Sampling Settings window 9:

*Adaptive Sampling* By selecting this option (recommended) an adaptive sampling method similar to the one described in [4] is used. In adaptive sampling the sampling distribution at the variables is adapted during the ongoing sampling process in order to achieve samples with higher importance weights. Adaptation is only performed for nodes with a maximal number of parents specified in the *Max CPT Parents* field. All other variables are sampled according to their initial distribution as specified by the relational Bayesian network. The number *Subsamples(Adaptation)* determines how quickly the initial distribution is adapted. Small numbers lead to a faster adaptation process.

*Subsamples(Min,Max)* determines the number of subsamples based on which the Min, Max, and Var values in the Query panel are computed.

*Sampling Order* can be set to 'forward' or 'ripple'. In forward mode variables are sampled in an ordering consistent with their dependency ordering, i.e. a variable is only sampled after all its parents have been sampled. In ripple mode, variables are sampled in an order starting with immediate neighbors of instantiated variables, and continuing outwards (both forward and backward with regard to the dependency ordering). Ripple mode can sometimes give better results when evidence is close to the leaves of the network. Ripple order cannot be combined with adaptive sampling.

By selecting options from the *Log* panel information on the sampling process can be logged: if a *Log file* is selected, then the information will be written to that file, otherwise it is written to standard output. *Trace* produces a log of the values displayed in the Query panel. A *Normal* trace only logs the P values, whereas a *Long* trace also logs the Min, Max and Var values. The *Sample Order* and the current *Evidence* can also be written to the logfile. *Network Stats* produces a short summary of network statistics, especially the number of nodes, distinguished by the number of their parents. The logfile is formatted so as to facillitate plotting with gnuplot the developement of P, Max, Min, and Var values with increasing sample size.

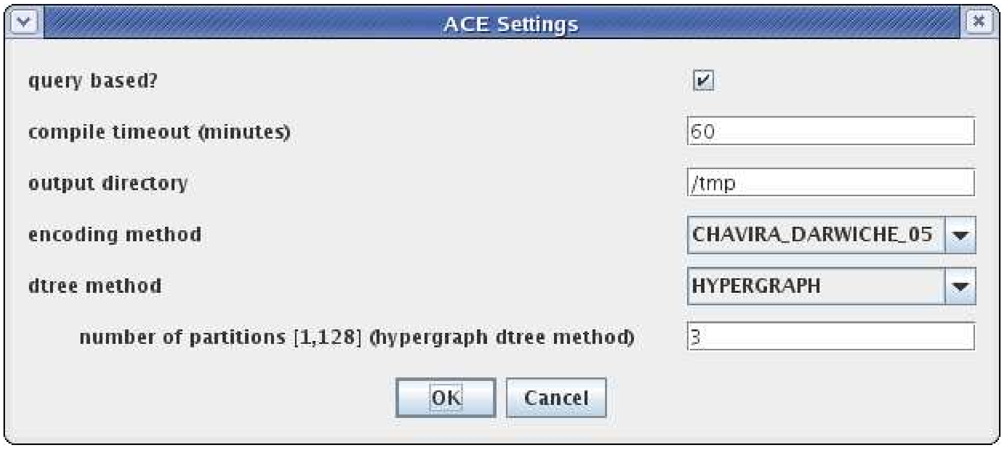
## ACE inference

The ACE inference module implements exact inference for Relational Bayesian Networks as described in [1]. This method performs a compilation into an *arithmetic circuit* of the Bayesian network defined by a Relational Bayesian Network and an input structure. As an intermediate step, ACE computes a logical representation in conjunctive normal form (CNF) of the *network polynomial*.

> *Note on using ACE:* To use the ACE inference methods the ACE system must be installed. ACE is available free at http://reasoning.cs.ucla.edu/ace/
> When first calling ACE inference from Primula, a file selection dialog will be opened in which the locations of the `ace.jar` and `inflib.jar` archive, as well as the `c2d` executable have to be specified.

When the "Compute" button of the ACE interface $\boxed{10}$ is pushed, the current model will be compiled into an arithmetic circuit, and the conditional probabilities of the query atoms given the current instantiation is displayed $\boxed{11}$. For complex models the compilation can take some time. The progress of the computation is indicated in the progress bar at the bottom of the ACE interface. After the initial compilation, the arithmetic circuit can be used for arbitrary queries: pushing the “Compute” button again after changing the instantiation and/or the query will update the probabilities in the ACE column. However, changes to the input structure will make a re-compilation necessary.

### Ace Settings



query based? ☑

compile timeout (minutes) 60

output directory /tmp

encoding method CHAVIRA_DARWICHE_05

dtree method HYPERGRAPH

number of partitions [1,128] (hypergraph dtree method) 3

*query based*: When checked, then compilation will generate an optimized arithmetic circuit for computing the probabilities of the current query atoms, given the current instantiation. This can be much faster than computing the full arithmetic circuit. However, changes to query and/or instantiation make a re-compilation necessary. Note that this Ace option is analogous to the conjunction of the options “Evidence conditioned” and “Query specific” in the Options menu of the Primula console. The setting in the Primula Options menu, however, only affects the Bayesian network construction in the Primula Run menu, not the Ace operations.

*compile timeout*: Timeout until giving up compilation…

*output directory*: Ace writes intermediate stages of the compilation into files. Here one specifies a directory that Ace can use for this temporary storage (e.g. /tmp on a Unix system).

*encoding method*: Specifies to use the network encodings according to [5], [15], [2], respectively [3]. See the Ace manual for further details on the different methods.

*dtree method*: Choose the heuristic for generating the decomposition tree of the CNF network encoding. See the ACE manual for more information on the different heuristics.

*number of partitions*: Parameter of the Hypertree heuristics for the decomposition tree construction. See the ACE manual for more information.

# 5 The Bavaria Relational Structure Editor

The Bavaria structure editor provides a graphical interface for the construction of relational input structures. The three lists on the left side of the window are used for the declaration of predefined relations. These are partitioned into ‘Attributes’ (unary relations), ‘Binary Relations’ and ‘Higher arities’ (relations of arity $\geq$ 3). The main difference between these three groups of predefined relations is the way they are graphically represented in the Bavaria window. Otherwise, the *Primula*system treats relations of all arities uniformly.

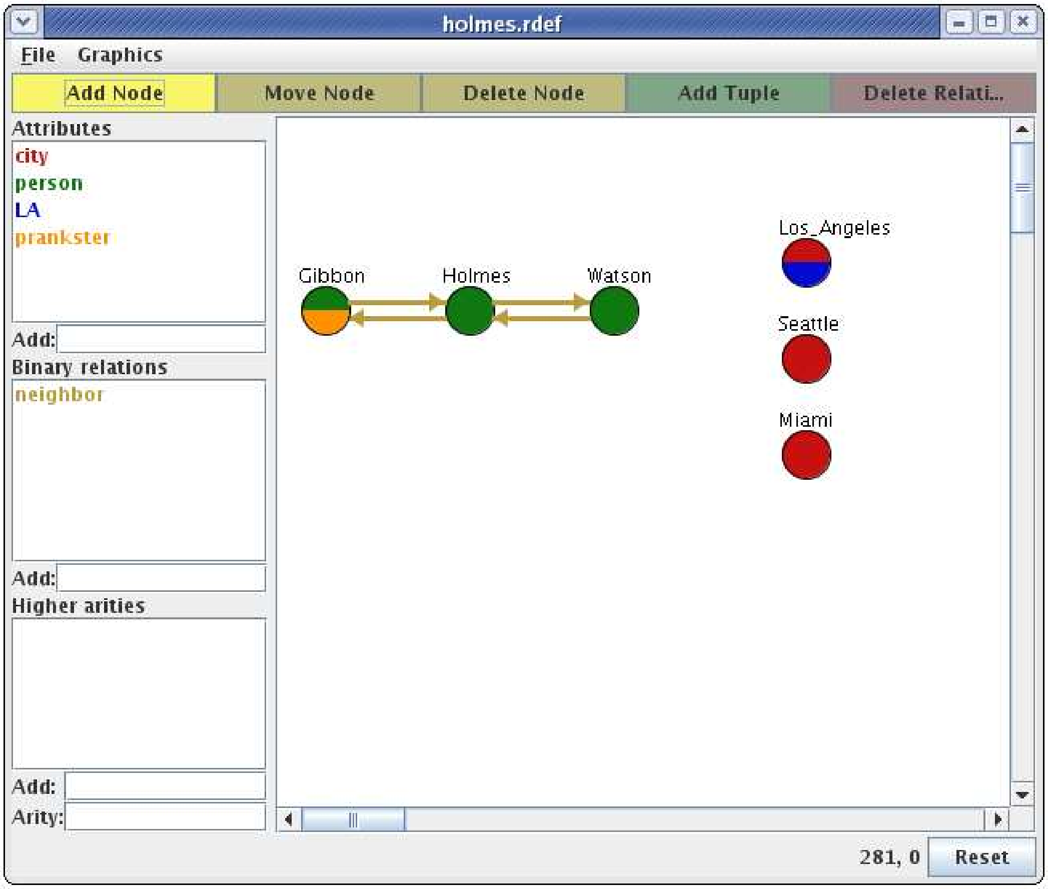


Figure 10: Bavaria Structure Editor

Each attribute and binary relation assigned a color. The initial default assignment can be edited by right-clicking on a relation name in the “Attributes” or “Binary Relations” lists. Each object that has one of the specified attributes is colored with the corresponding color. Similarly, binary relations are represented by colored arrows. Relations of higher arities are not graphically represented in the Bavaria window. They can nevertheless be defined and edited in Bavaria.

Depending on the chosen operating mode, the functions of the Bavaria editor are as follows:

**Add Node** Left mouse-click in the editing pane to create a new node (object).

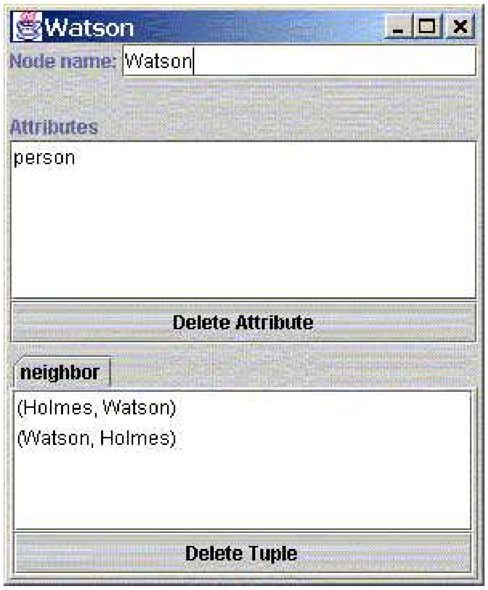
**Move Node** To move a node, select and drag with left mouse button.

**Delete Node** Delete node (and all the relation tuples it is part of) with left mouse-click.

**Add Tuple** Select one of the relations from either of the three relations lists. Add a tuple of objects to that relation by clicking on the objects in the editing pane. The incremental construction of tuple is echoed in the bottom line of the Bavaria window. Click on 'Reset' button to cancel current selection (only for binary and higher arities).

**Delete Relation** Deletes the specification of a whole relation by left mouse-click on the relation in the relations list.

Individual tuples in a relation are deleted using the *Node Window*. A node window can be opened in all operating modes of Bavaria by a right mouse-click on the node. The node window can be used to rename a node, to view all tuples of relations that this node is part of, and to delete any such tuple.



Node name: Watson

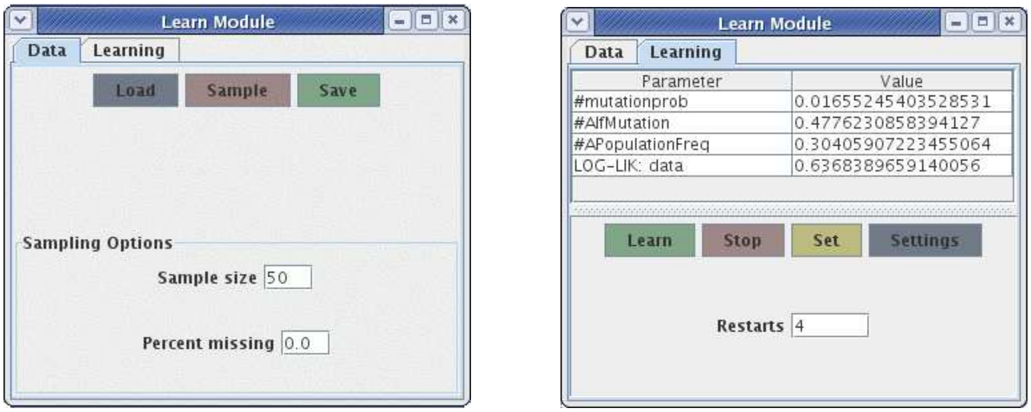
person

(Holmes, Watson)
(Watson, Holmes)

Figure 11: Node window

# 6 Parameter Learning

The parameters of a Relational Bayesian Network model can be learned from data using the gradient ascent method described in [12]. The learning is performed using the *Learn Module*, which is started from the *Primula*:*Run* menu.



Sample size 50

Percent missing 0.0

| Parameter | Value |
|---|---|
| #mutationprob | 0.01655245403528531 |
| #AIfMutation | 0.4776230858394127 |
| #APopulationFreq | 0.30405907223455064 |
| LOG-LIK: data | 0.6368389659140056 |

Restarts 4

Figure 12: The Learn Module

**Quick Demo: Learning (complete data)**

- Load the model `mendel.rbn` and the input domain `mendel.rdef`. Start the *Learn Module* from the *Primula Run* menu. Set e.g. *Sample size* = 50 in the *Learn Module* (press 'return' after entering the number into the textfield).
- Push the *Sample* button of the *Learn Module*. This will generate 50 complete instantiations of all the ground atoms defined by the probabilistic relations in `mendel.rbn` for the objects in `mendel.rdef`.
- Load `mendel_param.rbn` as *Model source*. In this file the salient probabilistic parameters of `mendel.rbn` have been replace by parameter variables *#mutationprob*, *#APopulationFreq*, and *#AIfMutation*.
- Push the *Learn* button in the *Learn Module:Learning* tab. The learner starts by building a *Gradient Graph* structure for computing the gradient of the likelihood function. This will take a few seconds; the progress is indicated in the *Primula* console. After completion of the *Gradient Graph*, a gradient ascent likelihood maximization is performed. Again after a few seconds, the values for the parameter variables in the model are displayed in the *Learn Module*.
- The learner will automatically restart the gradient ascent with different random initial parameter values. The number of restarts that have been performed so far is displayed in the *Learn Module*; the displayed parameter values are updated when a restart has resulted in parameter values with higher likelihood value than the previous ones.
- In this example it will be observed that the values for *#mutationprob* and *#APopulationFreq* are learned with a quite good accuracy when compared to the values in the generating model `mendel.rbn`, whereas *#AIfMutation* shows a relatively high variance. This is explained by the fact that *#AIfMutation* is a parameter in a mixture component with mixture coefficient 0.01, and therefore only has a relatively small impact on the generated data.

Table 2: Parameter learning demo

## Parametric RBN for learning

The parameter learning task for RBNs is to learn numeric parameters for an RBN model from data. In principle, every *Constant* in the syntax definition of RBNs is such a numeric parameter (cf. Appendix A). However, constants in an RBN can be of quite different nature: consider the `mendel.rbn` model. It contains the probability formula

```
AFather(v) =
      (@fatherInTree(v):
           (0.01:
                 0.6,
                 mean{AFather(u),AMother(u)|u: father(u,v)}
           ),
           0.3
      );
```

The three constants 0.01, 0.6, 0.3 in this formula represent stochastic parameters of the domain that one might want to learn from data. Now consider also from `mendel.rbn`:

```
Aa(v) = (AFather(v):(AMother(v):0,1),AMother(v));
```

This formula contains two constants 0 and 1. However, these constants do not represent truly stochastic parameters; they are needed to represent the deterministic knowledge that a person has genotype Aa if and only if exactly one of *AMother* and *AFather* is true.

In a *parametric RBN* some, not necessarily all, occurrences of constants are replaced by *parameter variables*, represented by strings starting with #. Parameter learning for RBNs will then determine values for the parameter variables. The same parameter variable can be used in several places in a parametric RBN, which provides a means of *parameter tying*, i.e. two different parameters are forced to be set to the same value.

`mendel_param` is an example for a parametric RBN. It contains the parameter variables *#mutationprob*, *#APopulationFreq*, and *#AIfMutation*. Each of these variables appears twice, once in the formula for *AFather*, and once in the formula for *AMother*. Thus, it is here assumed prior to learning that the inheritance of the A-allele follows the same laws in the maternal and paternal lines. The deterministic constants 0,1 in the formulas defining the genotypes have not been replaced by parameter variables.

## Data

The data for learning parameters in a given RBN consists of a collection of input domains for the predefined relations (often just one input domain), and for each input domain one or several observations (possibly incomplete) of the ground atoms constructible from probabilistic relations of the RBN and objects from the input domain.

```
AFather(v) =
      (@fatherInTree(v):
           (#mutationprob:
                 #AIfMutation,
                 mean{AFather(u),AMother(u)|u: father(u,v)}
           ),
           #APopulationFreq
      );

AMother(v) =
      (@motherInTree(v):
           (#mutationprob:
                 #AIfMutation,
                 mean{AFather(u),AMother(u)|u: mother(u,v)}
           ),
           #APopulationFreq
      );

AA(v) = (AFather(v):AMother(v),0);
Aa(v) = (AFather(v):(AMother(v):0,1),AMother(v));
aa(v) = (AFather(v):0,(AMother(v):0,1));
```

Table 3: Parametric RBN mendel_param.rbn

**Sampling data** When a RBN (without parameter variables) and an input domain are loaded, then a random sample for this model can be generated using the *Sample* function of the *Learn Module*. The *Sample size* specified in the *Learn Module* is the number of observations of the probabilistic atoms for the model and input domain. Given mendel.rbn as model and mendel.rdef as input domain, a single (complete) sample contains sampled truthvalues for $5 \cdot 39$ ground atoms constructible from the 5 (unary) probabilistic relations in the model, and the 39 objects in mendel.rdef. When *Percent missing* is set to $c > 0$, then $c\%$ of the sampled values are deleted (values are missing completely at random).

Sampled data is immediately set to be the ’active’ data used for learning. It can also be saved in .rdef format using the *Learn Module Save* function (see Appendix B for details on the .rdef format).

**Loading data from file** Using the *Load* button of the *Learn Module*, data in .rdef files can be loaded for learning. The learning data must be contained in a single file.

## Learning

After data has been loaded, the learning is controlled from the *Learn Module:Learning* tab. First, a parametric RBN whose predefined and probabilistic relations match the predefined and probabilistic relations of the dataset must be loaded as the *Primula:Model source*. Learning is then initiated by pressing the *Learn Module:Learn* button.

**Quick Demo: Learning (incomplete data)**

- As in Table 2 use model `mendel.rbn` and input domain `mendel.rdef`. Now sample 100 cases with 10% missing values.
- Load `mendel_softparam.rbn` as *Model source*. This is the same as `mendel_param.rbn`, but with relaxed probability formulas for the relations *AA*, *Aa*, and *aa*.
- Now push the *Learn* button in the *Learn Module:Learning* tab. The learning proceeds as before, but is considerably slower due to the sampling procedure (which dominates the time requirement). To monitor the progress of the optimization, check the *Verbose* option on the *Learn Module:Learning:Settings* panel.

Table 4: Parameter learning demo, continued

The learning algorithm consists of two phases: construction of the *Gradient Graph*, i.e. the computational structure used for evaluating the gradient of the likelihood function, and the actual gradient ascent optimization of the parameters. When the gradient ascent has terminated, the computed parameter values are displayed in the *Learn Module*, and the gradient ascent restarts with a new random initialization of the parameter values. Whenever a restarted gradient ascent terminates with parameter values with higher likelihood than the previous best values, the Parameter-Value table is updated. This process continues until the *Stop* button is pressed.

**Incomplete data** When data is incomplete, then the learning process also contains a routine for sampling values of unobserved ground atoms. Likelihood values and gradients needed in the gradient ascent are then estimated based on these sampled values (see Appendix sec:learnsample for details on the sampling process in gradient ascent). For the sampled values to be usable for the likelihood and gradient estimation, they must have non-zero probability given the model with the current parameter values, and the other instantiations in the training data.

Consider `mendel_param.rbn` as shown in Table 3. If, for example, several truth values for *AFather*, *AMother*, *AA*, *Aa*, and *aa* atoms need to be randomly sampled, then it will be extremely difficult to obtain a sample that does not violate some of the deterministic dependencies between these relations, and therefore has probability zero. The message *Failed to sample missing values* is printed on the Primula console when the sampler did not obtain samples with nonzero probability. To circumvent this problem, the best solution is to slightly relax the parametric model, so that no more atoms can have probability zero. The easiest way to do this, is to modify a probability formula $F(v)$ that may evaluate to zero (or one) into the relaxed form $(0.999 : F(v), 0.5)$, i.e. a mixture of the original $F(v)$ and the constant probability value $0.5$. This formula can no longer attain the values 0/1, and yet the change of the model is small enough so that the likelihood function of the parameters to be learned is not significantly different from the likelihood function defined by the original model.

**Setting the parameters and estimating the likelihood** The parameters in the parametric model are set to the learned parameter values by pressing the *Set* button in *Learn Module:Learning*. One can perform inference with this model, or save it to an .rbn file using *Primula:Run:Save RBN*.

When learning from incomplete data, the data likelihood displayed in the Parameter-Value table will tend to overestimate the true likelihood of the data, since this value is based on the same random sample of instantiations for the missing values as is used to estimate the parameters. An unbiased estimate of the likelihood is obtained by setting the learned parameters in the model, and pressing the *Learn* button again. When the loaded RBN model (no longer) contains unknown parameters, the learn function will just perform a likelihood computation of the data. Furthermore, restarts of the learning operation are now not used to mazimize the likelihood, but to improve the accuracy of the estimate. The value displayed after the termination of each restart is an estimate computed from all the samples in all the restarts. The likelihood value, therefore, may either increase or decrease after each restart.

**Learning Settings** Using *Learn Module:Learning:Settings* various parameters for the learning can be set:

- *Restarts*: Number of restarts of gradient ascent with initial random parameters. Default is -1, i.e. process is restarted open-ended until *Stop* button is pressed
- *Verbose*: Check this box to have a progress log printed to the system console. See also Appendix D
- *All other parameters* See Appendix D

# 7 Interpreting Markov Logic Networks

*Primula* also supports the language of *Markov Logic Networks (MLNs)* [14] for model specification and inference. MLNs are compiled into an RBN using the translation described in [9]. The translation is based on a MLN semantics that for MLN knowledge bases in conjunctive normal form (cnf) is equivalent to the one described in [14], and implemented in the *Alchemy* system (http://alchemy.cs.washington.edu/). The translation does not incorporate a cnf transformation of the original knowledge base (as specified by Table 2 in [14]).

As a result, *Alchemy* and *Primula* should yield the same results for queries on the cnf knowledge base

$$\begin{array}{l} 1\ \texttt{!smokes(x)vcancer(x)} \\ 1\ \texttt{smokes(x)v!cancer(x)} \end{array} \tag{1}$$

(cf. example file `friends_and_smokers_v3.mln`). However, for the non-cnf knowledge base

$$2\ (\texttt{smokes(x)} <=> \texttt{cancer(x)}) \tag{2}$$

(`friends_and_smokers_v2.mln`) the results differ: Under MLN/*Alchemy* semantics, knowledge base (2) is equivalent to (1), because (2) is first transformed into (1) before the features in the ground Markov network are generated. Thus, both (2) and (1) lead to a feature set where for each constant $a$ that can be substituted for $x$ there are two boolean features corresponding to the formulas !*smokes*$(a) \vee$ *cancer*$(a)$ and *smokes*$(a)\vee$!*cancer*$(a)$. The *Primula* semantics for MLN knowledge bases does not require a cnf-transformation, and (2) defines a distribution determined by one boolean feature *smokes*$(a) \Leftrightarrow$ *cancer*$(a)$ for each $a$.

**Quick Demo**

- Load `friends_and_smokers.mln` as *Primula*'s Model source.
  - when prompted to *Load domain data*, select `friends_and_smokers_domain.db`
  - when prompted to *Load evidence data*, select `friends_and_smokers_evidence.db`
- Start the *Primula* Inference Module, and set `smokes([person*])` and `cancer([person*])` as query.
- Run exact inference using ACE
- The results include $P(\textit{smokes}(\textit{Paul})) = 0.3794$, $P(\textit{cancer}(\textit{Bob})) = 0.5684$.

If *Alchemy* is available, run for comparison the query

```
 $ALCHEMYHOME/bin/infer
      -i friends_and_smokers.mln -r friends_and_smokers.result
      -e friends_and_smokers_domain.db,friends_and_smokers_evidence.db
      -cw friends
      -ow cancer,smokes
      -q smokes,cancer
      -ms -maxSteps 5000
```

This inference should yield (approximately) the same results as computed by *Primula* above (with higher settings of *maxSteps* leading to a better match of the results).

Table 5: Consistent inference in *Alchemy* and *Primula*

## 7.1 MLN inputs for *Primula*

*Primula* takes MLN inputs in the file formats specified for the *Alchemy* system[1]. These inputs are distributed over files in the .mln format containing the general model specification, and files in the .db format containing relational data. Roughly speaking, the .mln file contains a random relational structure model specified in the MLN language, whereas .db files will contain the specification of an input domain, as well as evidence on probabilistic relations.

The MLN interpreter of *Primula* implements the language of Markov Logic Networks as introduced in [14] in full generality. However, not all additional features available in *Alchemy* are implemented in *Primula*. The set of admissible .mln input files for *Primula*, therefore, is a subset of the admissible .mln files for *Alchemy*. Specifically, *Primula* accepts as input .mln files that have the form

```
Predicate declarations
Type declarations
Hard formulas
Weighted formulas
```

where:

[1] http://alchemy.cs.washington.edu/

*Predicate declarations* introduce the relations (predefined and probabilistic) with type specifications of their arguments. Example (we adopt the running example from [14]):

```
// predicate declarations
friends(person,person)
smokes(person)
```

Note that relation names have to start with lowercase letters.

*Type declarations* (optional) specify objects belonging to the different types. E.g.

```
//type declarations
person={Peter, Anna, Carl, Elisabeth}
```

Note that constants must start with uppercase letters.

*Hard formulas* are first-order formulas in the syntax specified by *Alchemy*, followed by a period ".". The syntax of first-order formulas is:

- Atomic formulas can be relational or equational atoms, e.g. `friends(x,y)`, `smokes(someperson)`. `someperson=x`. *Primula* only accepts .mln files in which all arguments of such atoms are variables (starting with lowercase letters), not constants. For example, `friends(Peter,y)` is not allowed in .mln input files for *Primula* (unlike *Alchemy*)[2].
- Boolean operators `!` (not), ^ (and), `v` (or), `=>` (implies), `<=>` (if and only if). Precedence of operators is in this order, i.e.

  ```
    r(x) v r(y) ^ !r(z) v s(x) => !s(y)
  ```

  is evaluated as

  ```
     (((r(x) v (r(y) ^ !r(z))) v s(x)) => !s(y))
  ```

- Quantifiers `exist` (or `Exist`, `EXIST`) and `forall` (or `Forall`, `FORALL`). Example:

  ```
  exist x,y (friend(x,y)^smoke(y))
  ```

  To ensure compatibility with the quantifier scoping rules of *Alchemy*, it is recommended that the sub-formula that is the intended scope of the quantifier is enclosed in parenthesis.

[2] this is not a real limitation of expressivity: a standard workaround would be to introduce a relation symbol `peter(person)`, to declare (in a .db file) that `peter(Peter)`, and then to replace `friends(Peter,y)` with `friends(z,y)`, where the variable `z` is relativized to the relation `peter`

No function symbols are allowed in input MLNs for *Primula*[3]. Furthermore, *Primula* does not support several syntactic extensions of the MLN framework that are provided by *Alchemy*, e.g. the + operator in front of variables.

*Weighted formulas* are first-order formulas preceded by a weight. Example:

```
1.1 friend(x,y) => (smokes(x) <=> smokes(y))
```

The basic definition of MLNs makes no distinction between predefined and probabilistic relations. For all relations that appear in the MLN knowledge base, (negated) ground atoms can be specified as evidence (usually collected in one or several data (.db) files). However, for inference, the *Alchemy* systems applies the *closed-world assumption* to some relations, i.e. all ground atoms of these closed-world relations that are not specified as *true* in the evidence, are considered to be *false*. See the *Alchemy* manual for details on which relations will be treated as closed-world. In our introductory example, the relation *friends* is declared closed-world by the command-line switch `-cw`, whereas the relations `smokes` and `cancer` are declared open-world. Closed-world relations are completely specified by the input, and therefore can be seen as predefined relations in the sense of *Primula*.

For *Primula*, the distinction between closed- and open-world relations is made by distributing ground evidence atoms over two input .db files: one, the *domain data file*, for atoms of closed-world relations, and one, the *evidence data file*, for atoms of open-world relations. No relation must appear both in the domain and the evidence data file. The domain data file should not contain any negated atoms. Together with the Type declarations in the .mln file, this yields the distinction between predefined and probabilistic relations, and the specification of the input domain:

- Predefined relations are:
  - Unary relations corresponding to the types specified in the .mln file
  - All relations for which there exists at least one ground atom in the domain data file.
- Probabilistic relations are
  - all other relations specified in the .mln file
  - new synthetic relations named *MLNRel0*,*MLNRel1*,... that are generated by the MLN to RBN compilation process.
- The input domain is constructed as follows:
  - The domain contains one object for each constant symbol appearing in the .mln file, or either of the two .db files.
  - A type relation $t$ is true for a constant $c$ iff $c$ is listed in the type declaration for $t$ in the .mln file, or $c$ appears in one of the .db files in a relation argument that according to the relation declaration in the .mln file is of type $t$.
  - A predefined relation $r$ is true for a tuple of constants $\boldsymbol{c}$ iff $r(\boldsymbol{c})$ is contained in the domain data file.

[3] Again, this is no limitation of expressivity if the *Known function assumption* [14] is made, which allows to replace function symbols by equivalent predefined relations

### 7.2 Performing Inference

After reading the MLN model and data files into *Primula*, inference can be performed using any of *Primula*'s inference methods. However, the main benefit of using *Primula* for MLN inference is to perform exact inference for small domains. The sampling inference in *Primula* is not as efficient for MLN models as the sampling methods provided by *Alchemy*.

In the inference module numerous *MLNRelx*-atoms will appear as instantiated to *true*. This is part of the translation MLN to RBN translation process, and necessary for representing in *Primula* the same distribution over the relations of interest as defined by the original MLN model.

While the input domain can be visualized in the *Bavaria* editor, it should not be modified in *Bavaria*: a modification of the input domain would require the generation and instantiation of additional *MLNRelx*-atoms, and an automatic update of the instantiated *MLNRelx*-atoms is not yet supported.

In the example in Table 5 the inference results obtained by *Alchemy* and *Primula* were consistent, because `friends_and_smokers.mln` is a cnf knowledge base. The issues with non-cnf knowledge bases are illustrated using the example inputs `friends_and_smokers_v2.mln` and `friends_and_smokers_evidence.db` The relation *friends* is now omitted, the domain consists of the single element *Anna* of type *person* (in the absence of predefined relations other than the type relations, just skip the loading of a domain data file by pushing the 'Cancel' button of the 'Load domain data' window), and the MLN knowledge base is just (2).

When querying *Primula* for the probability of *cancer(Anna)*, one obtains the result 0.1192. Running this query in *Alchemy*, on the other hand, gives a value of approximately 0.26, which corresponds to the result *Primula*gives for the query *cancer(Anna)* for inputs `friends_and_smokers_v3.mln` (i.e. knowledge base (1)) and `friends_and_smokers_evidence.db`.

# 8 The Elements of the Relational Bayesian Network Language

The *Primula* system reads probabilistic model specifications in the formal language defined in appendix A. This is essentially the language of Relational Bayesian Networks as introduced in [10]. In a Relational Bayesian Network, the distribution for a random relation is defined by a single declaration of the form

$$< \mathit{RAtom} >=< \mathit{ProbabilityFormula} >;$$

The language for the specification of Probability Formulas is defined by the four basic constructs *Constants*, *Indicators*, *Convex Combinations* and *Combination Functions*, and the two additional constructs *S-formulas* and *Macro-Reference* (these are additional in that they do not add to the expressivity of the language, but facillitate the model specification).

For an illustration of the first three basic constructs, table 6 shows the Relational Bayesian Network contained in `classic.rbn`. This is an RBN-encoding of a standard Bayesian network. It does not use any predefined relations, and defines three random attributes `a`,`b`,`c`. Line 1 contains the definition of the random attribute `a`, which is true for any object $v$ with probability 0.5. Line 2 defines the distributionof attribute `b` conditional on the truth value of attribute a using the convex combination construct

$$(< \textit{ProbabilityFormula} >:< \textit{ProbabilityFormula} >, < \textit{ProbabilityFormula} >)$$

In this case, the second and third probability formula arguments are just the constants 0.4 and 0.9,respectively. The first probability formula argument is the *Indicator* $\mathtt{a}(v)$. The semantics of the whole formula is that the probability of attribute b being true is 0.4 when attribute a is true, and 0.9 otherwise.

```
1 a(v)=0.5;
2 b(v)=(a(v):0.4,0.9);
3 c(v)= (a( v):
4        (b(v):0.2,0.4),
5        (b(v):0.1,0.3)
6     );
```

Table 6: A Standard Bayesian Network

Generally, probability formulas are evaluated by substituting specific objects $o$ for its free variables $v$. The indicator formula $\mathtt{a}(o)$ then evaluates to 1 if $\mathtt{a}(o)$ is true, and to 0 if $\mathtt{a}(o)$ is false. A convex combination with its three constituent probability formulas $PF_1$, $PF_2$, $PF_3$ evaluates to

$$\textit{eval}PF_1\textit{eval}PF_2 + (1 - \textit{eval}PF_1)\textit{eval}PF_3,$$

where $\textit{eval}PF_i$ are the (recursively computed) evaluations of its three constituent probability formulas.In the case where $PF_1$ is deterministic, i.e. always evaluates to either 0 or 1, this reduces to an if-then-else construct: the value of the convex combination is equal to $PF_2$ if $PF_1$ is 1 (”true”), and to $PF_3$ otherwise. In most of our examples we use convex combinations in this way, i.e. with deterministic first arguments.

Lines 3-6 define the probability distribution for c in very much the same way, only now nested convex combinations are used in a way that is equivalent to the specification of a conditional probability table for c given a and b.

When classic.rbn is instantiated with any relational input structure, it will generate one standard Bayesian network for the three attributes a,b,c for each object in the domain. The simplest possible input structure is classic.rdef consisting of only one object. See section 9.2 for an encoding using 0-ary relations.

The key to the expressivity of RBNs is the fourth construct for probability formulas, the *combination function*:

$$< \textit{CombinationFunction} > \{ < \textit{ProbFormList} > | < \textit{VarList} > : < \textit{SFormula} > \}$$

where

$$< \textit{CombinationFunction} >::= \mathtt{n-or}||\mathtt{mean}||\mathtt{invsum}$$

The combination function construct permits to define a probability value as a function of a set of probability values $\{p_1, \ldots, p_k\}$, which are given by recursively evaluating the probability formulas in ProbFormList. Currently the three different combination functions shown in table 7 are implemented in *Primula*.

A simple example for the use of combination functions is in line 5 of mendel.rbn (cf. table 1):

$$\mathtt{mean}\{\mathtt{AFather(u)}, \mathtt{AMother(u)}|\mathtt{u} : \mathtt{father(u, v)}\}$$

| Name | Syntax | Definition | Multilinear |
|---|---|---|---|
| noisy-or | `n-or` | $1-\prod_{i=1}^{k}(1-p_i)$ | yes |
| mean | `mean` | $(1/k)\sum_{i=1}^{k}p_i$ | yes |
| esum | `esum` | $\mathit{exp}(-\sum_{i=1}^{k}p_i)$ | no |
| inverse sum | `invsum` | $\mathit{min}\{1,1/\sum_{i=1}^{k}p_i\}$ | no |

Table 7: Combination functions in *Primula*

This formula contains the free variable $v$. It is evaluated under the substitution $v$=*Mike* by evaluating the probability formulas `AFather(u)` and `AMother(u)` for all $u$ for which `father`($u$, *Mike*) is true in the given input structure, and then computing the mean value of the obtained probabilities. According to the input structure of figure 3 we have to consider the substitution $u$=*Paul*. The whole formula then evaluates to 0 (=*mean*$\{0,0\}$) if `AFather`(*Paul*) and `AMother`(*Paul*) both are false, to 1/2 (=*mean*$\{1,0\}$) if exactly one is true, and to 1 (=`mean`$\{1,1\}$) if both are true.

In general, the probability formula arguments of a combination function are evaluated for all objects that make *SFormula* true when substituted for the variables in *VarList*. *SFormulas* are boolean combinations of relational atoms in the predefined relations. The syntax definition for *SFormulas* requires parentheses around all conjunctions and disjunctions (e.g. $((a\&b)\&c)$ must not be written as $(a\&b\&c)$).

Given some input structure, any ground *SFormula* evaluates to true or false, and thus can be used also as a deterministic probability formula. When used as a stand-alone probability formula (rather than as a syntactic component of a combination function), *SFormulas* must be prefixed with the keyword `sformula` and enclosed in parentheses.

This use of *SFormulas* together with a second illustration of combination functions is illustrated in `randgraph3.rbn`, which contains the single line

```
edge(v,w)=mean{sformula(u=v)|u:u=u};
```

This defines a classic (”sparse”) random graph model: according to this model, any two nodes $v, w$ in a graph of $n$ nodes are connected with probability $1/n$. To see how this model is represented by `randgraph3.rbn`, let $a, b$ be two nodes. To compute the probability of `edge`$(a, b)$ substitute $a, b$ for $v, w$ in the probability formula. As $w$ does not actually occur in the formula, we obtain `mean{sformula(u = `$a$` ) | u : u = u }`. We have to evaluate the probability formula `sformula(u = `$a$`)` for all nodes $c$ for which $c = c$ is true, i.e. for all $n$ nodes in the domain. This evaluation will return 0 for all $c \neq a$, and 1 for $c = a$. Thus, we finally apply mean to a set that contains $n-1$ zeros and one 1, which gives $1/n$.

The combination function *noisy-or* can be used for existential quantification: in the case where all $p_i$ are either 0 or 1, *noisy-or*$\{p_1, \ldots, p_k\}$ returns 1 iff there is at least one $p_i = 1$. This allows first-order quantification in the probabilistic modelling with probability formulas. An example for this use is the probability formula in line 1 of `mendel.rbn`:

```
n-or{sformula(father(u,v))|u:  u=u}
```

This evaluates to 1 for $v = a$ iff there exists some $b$ with *father*$(b, a)$.

The last syntactic element we have not discussed so far are *Macro Definitions*. A macro definition has the same syntactic form as the definition of a random relation, prefixed with the symbol @. The probability formula in the macro definition can then be referenced by the macro's name. See table 1 for an example. When a macro is referenced, variable substitution is performed, i.e. the variables in the macro definition need not be the same as the variables in the macro reference.

*Note*: A macro definition must precede any reference to that macro in the input file.

# 9 Miscellaneous Issues

## 9.1 Relation Argument Types

In *Primula* any *unary predefined relation* can be used as a type constraint on the arguments of a probabilistic relation. Syntactically, type constraints are specified by preceding the arguments in the probabilistic relation definitions by the name of the type relations in brackets; e.g. in `distributed.rbn`:

```
hasmessage([agent]v,[time]t) = ...
```

If no type constraint is given for a relation argument, then the argument is assumed to range over all objects in the domain. For example, in `mendel.rbn` arguments are not explicitly typed, since all relation-arguments in this model range over all objects of the domain (for which no explicity 'person' type needs to be introduced).

By combining type constraints with unary relations describing unique objects, one can specify probabilistic attributes for specific objects. For example, in `holmes.rbn` the specification

```
earthquake([LA]w)=0.01;
```

turns 'earthquake' into an attribute for the specific object 'LA' (assuming that as intended in this model, the extension of the relation 'LA' in the input domain consists of a single city object).

## 9.2 Arity-0 probabilistic relations

It is also allowed to specify probabilistic relations of arity 0 in an .rbn file. For example, the following is a valid specification:

```
exists_infected() = n-or{ infected(v)| v:person(v)};
```

Probabilistic relations of arity 0 represent global properties of a random relational structure (here the existence of at least one infected person – a more interesting example is contained in `colorgraph.rbn`).

Independent of the size of the input structure, they give rise to exactly one random variable. Another example for the use of 0-ary relations is `classic_arity0.rbn`, which encodes a standard 3-node Bayesian network (cf. Section 8) using 0-ary relations. Regardless of the input domain, this model will always yield a Bayesain network with just 3 nodes.

Note that *Primula* only permits probabilistic relations of arity 0, not predefined relations of arity 0. The latter don't pose any problems in principle, but have not been implemented so far.

### 9.3 The Use of Predefined Order Relations

On all relational input structures predefined order relations *less,pred,zero,last* are automatically given.The user has only indirect control over *which* order is imposed on the objects of the structure (when the structure is constructed in the Bavaria editor, it will be the order in which objects were added to the domain). As a result, these order relations should be considered as a purely arbitrary ordering of the domain, with no semantic meaning. Consequently, these order relations are safely used in probabilityformulas only when no assumptioons on the special form of the order are made, and the distribution of the probabilistic relations remains the same when the underlying order changes. An example for such a use of the predefined order relations is in the first part of `randblock.rbn`:

```
@predselected(a,b)=n-or{blocks(a,c)|c: less(c,b)};

@selprob(v)=mean{sformula(z=v)|z: (less(v,z)|v=z)};

@selects(a,b)=(  @  predselected(a,b):
                   0,@selprob(b)
                  );

 blocks(u,v)=(  sformula((block(u) & location(v))):
                    @selects(u,v),0
                 );
```

Here the probability that a block $u$ is placed on location $v$ is specified sequentially for all $v$ according to the predefined order *less*: if $u$ was already placed on a location preceding the current one in the order, i.e. if the macro `@predselected(`$u$`, `$v$`)` evaluates to 1, then this probability is 0. Otherwise the probability is computed with the macro `@selprob(`$v$`)`, which returns $1/m$, where $m$ is the number of objects still following $v$ in the order. Note that the final result of this definition is that every block selects every location with equal probability, and that this remains the same when the underlying order *less* is permuted.

An example where this invariance under permutations of the predefined order relations does not hold is `dynamic.rbn`: here the probability of `a`$(v)$ and `b`$(v)$ depends on the position of $v$ in the ordering. One could specify with the Bavaria editor two seemingly identical input structures that would instantiate `dynamic.rbn` to two different models (because the nodes in the structure were added in a different order,and therefore the invisible order relations are different).

## 9.4 Intended, Valid and Invalid Input Structures

The probabilistic models encoded by a RBN are almost always created for a specific class of intended input structures. The inheritance model in table 1, for instance, is meant to be instantiated with input structures where every individual has at most one father and one mother. However, one can also instantiate the model with an input structure that does not satisfy this condition: one may add, for instance, in the input structure shown in figure 3 another *father*-edge from Susan to Mike. When *Primula* is run on this input structure, it will generate a Bayesian network that represents a probability distribution over the probabilistic relations. The new input structure is valid for the probabilistic model, even though the resulting distribution may not be very meaningful.

An input structure only is *invalid* for a given RBN specification, if it induces cyclic dependencies between the atoms of the probabilistic relations, and therefore there is no well-defined probability distribution. An example would be the input structure obtained from figure 3 by inserting a *father* edge from Katie to Mary. When run on this input structure *Primula* will respond with an error message ”Network is cyclic” (where network refers to the Bayesian network *Primula* attempted to construct - very often cycles in the Bayesian network will be induced by cyclic structures in the input structure, but there also can be other reasons).

## 9.5 Graph layout

The *Primula* system contains a graph-drawing algorithm based on [16] for computing the layout of the generated Bayesian networks. This layout is usually far from perfect. The layouts shown in the figures in this manual were obtained by manually editing the *Primula*-generated layouts.

# A Syntax

Following is a syntax specification of .rbn model files in extended Backus-Naur form.

```
< RelationalBayesianNetwork > ::= {< MacroDefinition > ";"| < RandomRelationDefinition > ";"};
< MacroDefinition > ::= "@" < MAtom > "=" < ProbabilityFormula >;
< RandomRelationDefinition > ::= < DefRAtom > "=" < ProbabilityFormula >;
< ProbabilityFormula > ::=
        < Constant >
        | < RAtom >
        |"(" < ProbabilityFormula > ":" < ProbabilityFormula > "," < ProbabilityFormula > ")"
        | < CombinationFunction > "{" < ProbFormList > "|" < VarList > ":" < SFormula > "}"
        |"sformula(" < SFormula > ")"
        |"@" < MAtom >;
< CombinationFunction > ::= "n − or" |"mean" |"invsum" |"esum";
< ProbFormList > ::= < ProbabilityFormula > {"," < ProbabilityFormula >};
< VarList > ::= < VarName > {"," < VarName >};
< SFormula > ::=
        < SAtom >
        |"~" < SFormula >
        |"(" < SFormula > "&" < SFormula > ")"
        |"(" < SFormula > "|" < SFormula > ")";
< DefRAtom >=< RRelationName >< TypedRRelationArguments >;
< MAtom >=< MRelationName >< RelationArguments >;
< SAtom >=< SRelationName >< RelationArguments >;
< RAtom >=< RRelationName >< RRelationArguments >;
< MRelationName > ::= < RelationName >;
< SRelationName > ::= < RelationName >;
< RRelationName > ::= < RelationName >;
< RelationName > ::= "a" | ... |"Z" {"a" | ... |"Z" |"0" | ... |"9" |"_" |"—"};
< MRelationArguments > ::= < EmptyVarList > | < NonEmptyVarList >;
< SRelationArguments > ::= < NonEmptyVarList >;
< RRelationArguments > ::= < EmptyVarList > | < NonEmptyVarList >;
< TypedRRelationArguments >::= < EmptyVarList > | < NonEmptyTypedVarList >;
< EmptyVarList >::= "()";
< NonEmptyVarList >::= "(" < VarName > {, < VarName >}")";
< NonEmptyTypedVarList >::= "(" < TypedVarName > {, < TypedVarName >}")";
< TypedVarName >::= {< TypeSpec >} < VarName >;
< TypeSpec >::= "[" < SRelationName > "]";
< VarName > ::= "a" | ... |"z" {"a" | ... |"z" |"0" | ... |"9"};
< Constant > ::= "0" |"1" |"0."{"0" | ... |"9"};
```

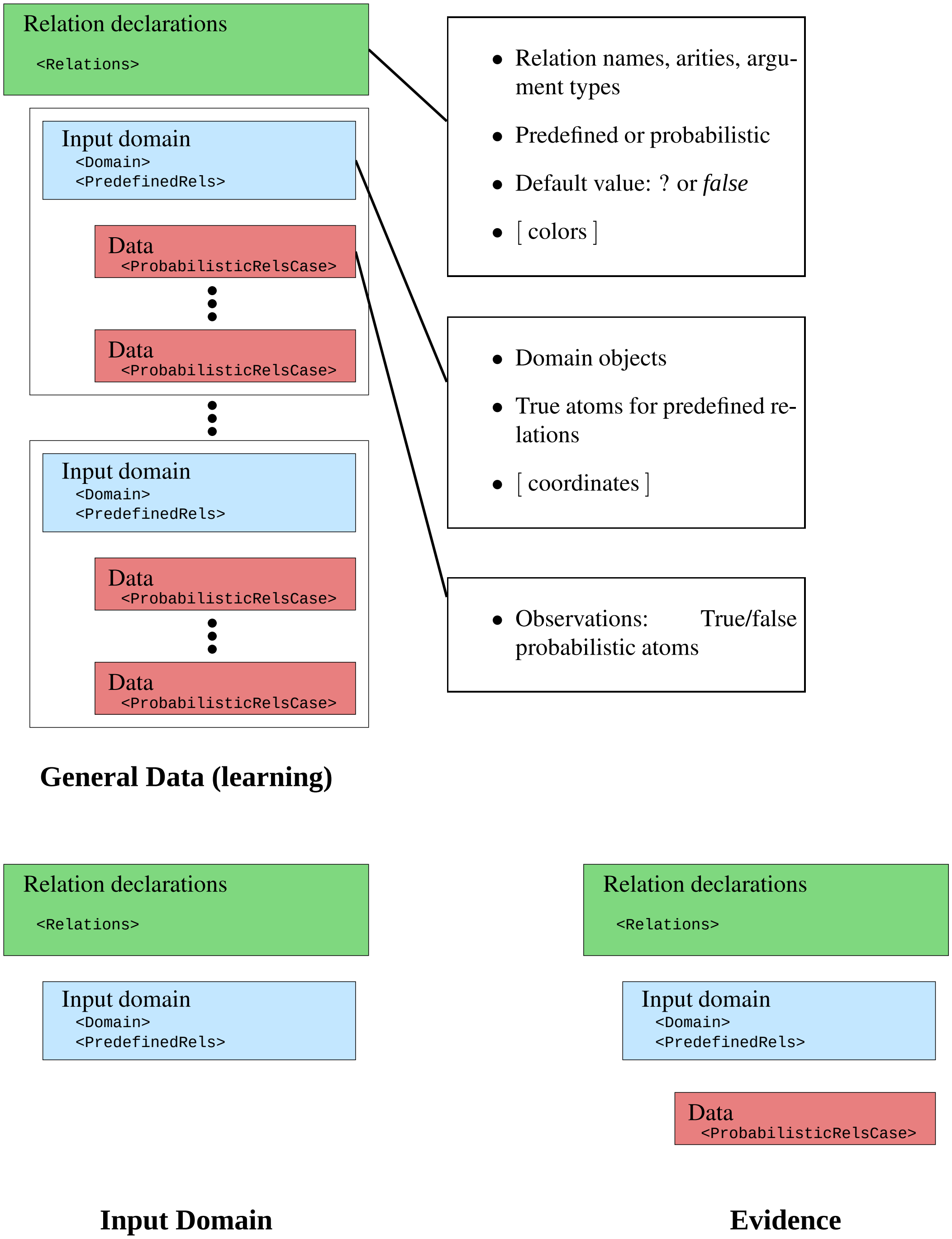
An input file may contain comment-lines starting with %. The only syntactic difference between *MAtom*, *SAtom* and *RAtom* is that *MAtoms* and *RAtoms* may have an empty argument list. Another reason for specifying these three kinds of atoms separately in the syntax is to emphasize the fact that the basic atom construct is used in three semantically distinct ways, and that one should chose distinct names for the three types of relations.

## B .rdef File Format

The .rdef format is a rich xml file format for relational data. It is used by *Primula* to save input domains, sample data for learning, and instantiation (evidence) data for inference.

Figure 13 shows the general structure of the data specification in an .rdef file. It starts with a header enclosed by `<Relations>` tags declaring all the relations appearing in the data, distinguished as predefined or probabilistic. Relations have an attribute `default` which specifies whether the value of ground atoms of that relation when not given in the data should be considered as *false* (i.e. the closed-world assumption is made for that relation), or as unknown.

Following the relation declarations is a sequence of blocks that are enclosed by `<DataForInputDomain>` tags. Each such block of data represents the observation of a single input domain, and possibly multiple observations (enclosed by `<ProbabilisticRelsCase>` tags) of the probabilistic relations associated with that input domain.



**General Data (learning)**

**Input Domain**

**Evidence**

Figure 13: Structure of .rdef files

*Primula*'s learn module can read and learn from data of the general structure shown in Figure 13. However, the sampling facility of of the learn module only can sample from one input structure at a time, so that data files generated by the sampler will only contain one `<DataForInputDomain>` block.

Data files representing input domains or evidence have the special structure shown in Figure 13.

### B.1 Reading Prolog Data

Using the Primula 'Console:Run:Convert Relational Data' function one can transform a database in Prolog format into rdef.

It is assumed that the input file consists of a list of ground relational atoms, each atom terminated by a period.

Since the rdef format can represent more information than a Prolog database (e.g. distinction of predefined/probabilistic relations, default values, argument types, etc.), an interactive 'upgrade' dialogue is used to enrich the source data. Currently, this dialogue only supports the definition of argument types: every unary relation appearing in the data is considered to be a candidate type predicate. The user can select from a list of all such unary relations those that are to be considered as type relations. Subsequently, the arguments of all other relations are assigned a type, if the data contains an atom with an argument belonging to that type.

Example: the Prolog data contains (among otheres) the ground atoms `person(tom)`,`divorced(mary)`, `owns(tom,account_427645)`. The user will be asked to select type predicates from the candidates *person* and *divorced*. If the user selects *person* as the only type predicate, then the relation *owns* will be assigned type *person* for its first argument. The second argument of *owns*, as well as the single argument of *divorced* will have the default type *domain* that applies to all objects in the domain. The system does not try to resolve conflicts: if the data contains also `male(tom)`, and the user also specifies *male* as a type argument, then the type of the first argument of *owns* will be an essentially random choice between *person* and *male* (in reality depending on the order in which atoms and relations first appear in the data).

Currently there is no support for an interactive specification of predefined or probabilistic relations. All declared type relations are interpreted as predefined, all others as probabilistic.

## C .rst File Format

The .rst file format is a simpler file format for storing input domains. It was used until *Primula*V. 2.1, and is still supported for compatibility.

An .rst file contains one *domain* declaration, one *coordinates* declaration, and a number of *relation* declaration.

A domain declaration has the format

`DOMAIN:` { <*comma-separated list of object names*> }`;`

Valid object names are arbitrary alpha-numeric strings that also may contain the special characters . or - or _.

A coordinates declaration has the format

`COORDINATES:` { <*list of x,y-coordinates*> }`;`

An x,y-coordinate has the form $(i, j)$ with integers $i, j$.

A relation declaration has the format

`Relation:` < *SRelationName*> `/` <*arity*> { <*list of tuples*> }`;`

Here *SRelationName* is as in the preceding syntax definition, *arity* is the arity of the relation specified. The list of tuples enumerates all tuples of domain objects that belong to the specified relation. Objects here are referenced by their internal integer representation, which is equal to the position of the object in the domain declaration list (counting starts with 0). If *arity*$=k$, then each tuple has the format $(i_1, \ldots, i_k)$. The parentheses can be omitted if tuples are separated by white space (there must be no white space within a tuple specification).

# D Details on Gradient Ascent and Sampling

Parameters of the learning procedure that can be set in the *Learning Settings* menu are printed in bold.

The gradient ascent parameter learning process proceeds in two phases.

In the first phase, a basic iteration starting at current parameter values $\boldsymbol{\theta}^{(t)} = (\theta_1^{(t)}, \ldots, \theta_K^{(t)}$ ($K$ the number of parameter variables to be learned) consists of

- computing (approximately) the gradient of the likelihood function at $\boldsymbol{\theta}^{(t)}$.
- performing a *linesearch* in the direction of the gradient. This linesearch is essentially a binary search for a local maximum of the likelihood function in the direction of the gradient. Only likelihood values, no first or second derivatives are used in this search.
- repeat with $\boldsymbol{\theta}^{(t+1)}$: local maximum found in the linesearch

These iterations continue until

$$|\boldsymbol{\theta}^{(t+1)} - \boldsymbol{\theta}^{(t)}| < \textbf{Distance threshold},$$

or

$$t > \textbf{Max. iterations gradient search}.$$

It can happen that after the this procedure has terminated, some parameters are (approximately) correctly estimated (assuming the generating model is available for verification), whereas others are not. This will typically happen when the number of ground atoms that depend on these different parameters greatly differs. Consider, for example, the very simple parametric RBN

```
t(v)=#param1;
r(v,w)=#param2;
```

A random sample for an input domain with 100 objects will then contain 100 ground *t*=atoms, but 10000 ground *r*-atoms, i.e. the likelihood function is a product of 100 terms depending on *#param1*, and 10000 terms depending on *#param2*. Thus, the value of *#param2* dominates the likelihood value, and the gradient ascent will terminate when it has found a (local) maximum for *#param2*. For this reason, the gradient ascent procedure is extended with a second phase, where the likelihood function is optimized individually for each parameter in turn. A basic iteration of the second phase is:

For $i = 1, \ldots, K$:

- compute (approximately) the partial derivative of the likelihood function at $\boldsymbol{\theta}^{(t)}$ with respect to $\theta_i$ (in fact, only the sign of the derivative is needed)
- perform a *linesearch* for an optimal value of $\theta_i$.
- continue with $\boldsymbol{\theta}^{(t+1)}$: local maximum found in the linesearch

When data is incomplete, then both the computation of the gradient, and the computation of likelihood values in the linesearch are based on a random sample of instantiations of the missing values. The random samples are obtained by Gibbs sampling. During the whole gradient ascent process, **Gibbs chains** many Markov chains for Gibbs sampling are maintained. One round of Gibbs sampling (i.e. one resampling of each ground atom atom with missing value) is performed at the termination of every linesearch in either of the two phases of the gradient ascent. Computations of gradients and likelihood values are based on the **Gibbs Window Size** many recent samples in each chain.

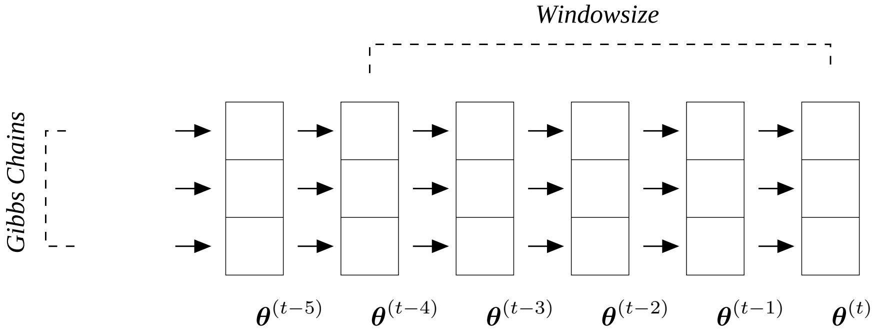

Windowsize
Gibbs Chains
$\boldsymbol{\theta}^{(t-5)}$
$\boldsymbol{\theta}^{(t-4)}$
$\boldsymbol{\theta}^{(t-3)}$
$\boldsymbol{\theta}^{(t-2)}$
$\boldsymbol{\theta}^{(t-1)}$
$\boldsymbol{\theta}^{(t)}$

Figure D illustrates the sampling process. Each box represents one instantiation of all unobserved ground atoms. In this figure **Gibbs chains** = 3 and **Gibbs Window Size** = 5. Thus, for the (approximate) computation of the gradient at $\boldsymbol{\theta}^{(t)}$ alltogether 15 complete instantiations of the unobserved atoms are used. Note that, ideally, all samples used for the computations at $\boldsymbol{\theta}^{(t)}$ should be obtained from Gibbs-sampling using $\boldsymbol{\theta}^{(t)}$. Here, for efficiency reasons, we also re-use samples obtained using previous parameters $\boldsymbol{\theta}^{(t-i)}$.

*Max. iterations gradient search* Max. iterations per phase.

*Distance threshold* Euclidean distance (linesearch and main loop)

*Likelihood threshold (linesearch)* $1 - Lt < ratio < 1 + Lt$.

*Max. fails (Sample Missing)* Overall $Mf \cdot numchains$ samples of prob. 0 are allowed. If exhausted, restarts with new random parameters.

# E Known Bugs

Some graphics components can reach an unstable state in which they generate exceptions like:

```
Exception in thread "AWT-EventQueue-0" java.lang.NullPointerException
        at javax.swing.plaf.basic.BasicProgressBarUI
           .updateSizes(BasicProgressBa rUI.java:470)
```

This can be annoying, but has no effect on the functionality of the system.

An error

```
warning: RunAceCompile.run() caught java.lang.Exception: Error reading
network!
```

occurs when Ace produces a Bayesian network file in .net format (Hugin) that does not comply with the Hugin naming conventions. Ace inference synthesizes filenames from the input .rbn and .rst files. An illegal name is generated when any of these files contains characters not admissible for Hugin .net files (e.g. numbers). Solution: use for .rbn and .rst files names containing only characters a-z,A-Z, and underscore _

# Acknowledgments

Jani Saijos, Mads Thrane, Keith Cascio, Mark Chavira and Alberto García Collado made important contributions to *Primula*'s development. Their help is gratefully acknowledged! *Primula* uses the *dom4j* XML parser (www.dom4j.org) – thanks also to the dom4j project for providing it!

# References

[1] M. Chavira, A. Darwiche, and M. Jaeger. Compiling relational bayesian networks for exact inference. *Int. J. of Approximate Reasoning*, 42:4–20, 2006.

[2] Mark Chavira and Adnan Darwiche. Compiling Bayesian networks with local structure. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1306–1312, 2005.

[3] Mark Chavira and Adnan Darwiche. Encoding cnfs to empower component analysis. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 61–74. Springer Berlin / Heidelberg, Lecture Notes in Computer Science, Volume 4121, 2006.

[4] J. Cheng and M. Druzdzel. AIS-BN: An adaptive importance sampling algorithm for evidential reasoning in large bayesian networks. *J. of Artificial Intelligence Research*, 13, 2000.

[5] Adnan Darwiche. A logical approach to factoring belief networks. In *Proceedings of KR*, pages 409–420, 2002.

[6] M. Fishelson and D. Geiger. Exact genetic linkage computations for general pedigrees. *Bioinformatics*, 18(Suppl. 1):S189–S198, 2002.

[7] N. Friedman, Lise Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, 1999.

[8] L. Getoor, N. Friedman, D. Koller, and B. Taskar. Learning probabilistic models of relational structure. In *Proceedings of the 18th International Conference on Machine Learning*, pages 170–177, 2001.

[9] M. Jaeger. Model-theoretic expressivity analysis. In L. De Raedt, K. Frasconi, P.and Kersting, and S.H. Muggleton, editors, *Probabilistic Inductive Logic Programming*, volume 4911 of *LNCS*, pages 325–339. Springer, 2008.

[10] M. Jaeger. Relational bayesian networks. In Dan Geiger and Prakash Pundalik Shenoy, editors, *Proceedings of the 13th Conference of Uncertainty in Artificial Intelligence (UAI-13)*, pages 266–273, Providence, USA, 1997. Morgan Kaufmann.

[11] M. Jaeger. Complex probabilistic modeling with recursive relational Bayesian networks. *Annals of Mathematics and Artificial Intelligence*, 32:179–220, 2001.

[12] M. Jaeger. Parameter learning for relational Bayesian networks. In *Proceedings of the 24th International Conference on Machine Learning (ICML)*, 2007.

[13] H. Pasula and S. Russell. Approximate inference for first-order probabilistic languages. In *Proceedings of IJCAI-01*, pages 741–748, 2001.

[14] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1-2):107 – 136, 2006.

[15] Tian Sang, Paul Beame, and Henry Kautz. Solving Bayesian networks by weighted model counting. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, volume 1, pages 475–482. AAAI Press, 2005.

[16] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Trans. on Systems, Man, and Cybernetics*, SMC-11(2):109–125, 1981.